FACULTADE DE MATEMÁTICAS

Traballo Fin de Grao

Introdución ás redes neuronais

Ana Martínez Leboráns

Xullo, 2022

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

GRAO DE MATEMÁTICAS

Traballo Fin de Grao

Introdución ás redes neuronais

Ana Martínez Leboráns

Xullo, 2022

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Traballo proposto

Área de Coñecemento: Estatística e Investigación Operativa

Título: Introdución ás Redes Neuronais

Breve descrición do contido

A aprendizaxe automática é un amplo campo dentro da ciencia que abarca un gran número de modelos e técnicas destinados a extraer información de certos conxuntos de datos. Entre estes modelos se atopan as redes neuronais, cuxos fundamentos matemáticos constitúen o obxecto de estudo deste traballo. O modelo de redes neuronais ten unha grande variedade de aplicacións, como por exemplo a monitorización de procesos sanitarios, entre outras moitas. Neste traballo, presentaranse conceptos como o de perceptron, o propio de rede neuronal, as cuestións relacionadas co adestramento da mesma e o algoritmo de propagación inversa. Tamén se abordará a presentación dalgunhas das opcións existentes dende o campo da optimización, encamiñadas a incrementar o rendemento dunha rede neuronal. Finalmente, diferentes exemplos de aplicación ilustrarán e completarán os conceptos e algoritmos presentados.

Índice

Resumo **VIII**

Introdución **XI**

1. Fundamentos **1**

1.1. Contexto histórico 1

1.1.1. *Machine learning* 2

1.1.2. *Deep learning* 3

1.1.3. Introdución histórica 5

1.2. Coñecementos previos 8

1.2.1. Sobreaxuste e subaxuste 9

2. Modelos e algoritmos **11**

2.1. Funcións de activación 11

2.2. Tipos de redes 14

2.2.1. Perceptrón 14

2.2.2. Redes de propagación cara diante 14

2.2.3. Redes recorrentes 15

2.2.4. Redes convolucionais 20

2.3. Funcións de custo e perda 27

2.4. Algoritmos de aprendizaxe 29

2.4.1. Descenso de gradiente 29
2.4.2. Retropropagación 32
2.4.3. Métodos de optimización 34
2.5. Hiperparámetros 38
2.5.1. Optimización de hiperparámetros 38

3. Aplicacións **41**
3.1. Exemplo en Python 41
3.2. Aplicacións reais 44

Anexo I: Código **49**

Bibliografía **55**

Resumo

A aprendizaxe profunda é unha disciplina que está en auxe dada a súa adecuación para resolver problemas complexos. Este traballo recolle a historia da aprendizaxe profunda dende os seus inicios, así como os conceptos básicos necesarios para a súa comprensión. Introdúcense as definicións formais dalgunhas funcións de activación, distintos tipos de redes neuronais como as redes recorrentes ou convolucionais e algoritmos e conceptos relativos ao seu adestramento. Finalmente, amósanse algunhas das aplicacións que ten no mundo real.

Abstract

Deep learning is a discipline that is on the rise given its suitability for solving complex problems. This project encompases the history of deep learning from its beginnings, as well as the basic concepts needed for its understanding. It introduces the formal definitions of some activation functions, several types of neural networks such as recurrent or convolutional networks and algorithms and concepts related to their training. Finally, it showcases some of the applications it has on the real world.

Introdución

A intelixencia artificial (IA) é un campo da informática moi importante e activo, con moitas aplicacións prácticas pero do que aínda queda moito por descubrir. Dentro deste gran campo está o *machine learning* (ML), ou aprendizaxe automática[1], que pretende que os ordenadores comprendan conceptos subxectivos e complexos por medio de recoñecemento de patróns, e non baseándose nunhas regras dadas polos programadores. O *deep learning* (DL), ou aprendizaxe profunda[2], é un tipo de aprendizaxe automática que se basea en utilizar moitas capas para representar unha xerarquía de conceptos, dende os máis simples ata outros máis complicados, e a familia de algoritmos que utiliza son chamados redes neuronais, que trataremos en detalle neste traballo.

No primeiro capítulo deste Traballo de Fin de Grao (TFG) presentaranse o contexto histórico e fundamentos necesarios para poder comprender a aprendizaxe automática, indicando certas recomendacións de lectura para complementar os conceptos presentados. A continuación, no segundo capítulo introduciranse os conceptos básicos necesarios para a utilización das redes neuronais. Isto inclúe os conceptos de función de activación, perceptrón, os principais tipos de redes existentes, definicións de funcións de custo e perda, algoritmos de aprendizaxe e os seus métodos de optimización e o concepto de hiperparámetros e a súa optimización. Finalmente, no terceiro capítulo presentaranse un exemplo sinxelo implementado en Python de tres modelos de redes neuronais, comparando os resultados obtidos con cada un deles, e diferentes aplicacións reais do estado do arte de aprendizaxe profunda.

[1]Empregaranse os termos *machine learning* e aprendizaxe automática indistintamente.
[2]Empregaranse os termos *deep learning* e aprendizaxe profunda indistintamente.

Capítulo 1

Fundamentos

Neste traballo realízase unha introdución ás redes neuronais. Trátase de modelos adecuados para a resolución de problemas que xorden en campos como a aprendizaxe automática ou a aprendizaxe profunda. Por iso, comezamos realizando unha presentación preliminar das mesmas e un percorrido a través dos seus principais fitos históricos.

Para este capítulo empregouse como referencia principal o libro de Goodfellow *et al.* (2016).

1.1. Contexto histórico

Dende os tempos da antiga Grecia, os seres humanos soñaban con crear máquinas que puideran pensar. Cando se concibiu por primeira vez a idea dos ordenadores programables, a xente xa se preguntaba se esas máquinas tornarían intelixentes, máis de cen anos antes de que Charles Babbage puidera sequera construír o primeiro deles, en 1822 (Swade, 2000).

A **intelixencia artificial** refírese a sistemas que pretenden imitar a intelixencia humana para resolver tarefas e poden mellorar iterativamente coa información que recollen, é dicir, que poden aprender. Nos inicios da intelixencia artificial descubriuse que se podían resolver moi rapidamente e de maneira moi sinxela problemas baseados en regras matemáticas que para os humanos resultaban intelectualmente complexos. Un exemplo disto é a máquina *Deep Blue* de IBM, que conseguiu vencer ao campión mundial de xadrez Garry Kasparov en 1997. Isto débese a que podemos resumir o xadrez en regras matemáticas concretas, cun número de espazos, pezas e movementos limitados para cada unha delas (Goodfellow *et al.*, 2016).

Porén, o problema estaba en resolver tarefas que para os humanos son intuitivamente fáciles pero para as que non é sinxelo dar descricións formais, como o recoñecemento de obxectos en imaxes. Para isto se precisa de moito coñecemento sobre o mundo, do cal unha gran parte é

subxectivo e intuitivo, que utilizamos nas accións do noso día a día.

Varios proxectos trataron de codificar estes coñecementos directamente no ordenador con regras de inferencia lóxica (coñecido como enfoque baseado no coñecemento), como é o caso de Cyc. Este proxecto, iniciado en 1984 por Doug Lenat, pretendía proporcionarlle ao ordenador o que denominamos como "sentido común" (Panton *et al.*, 2006). Os supervisores humanos introducían sentencias lóxicas ao sistema na linguaxe CycL, que consideraba 5 posibles estados das mesmas (monotonamente falsa/verdadeira, falsa/verdadeira predeterminada ou descoñecida). O maior problema estaba en escribir sentencias formais con suficiente complexidade para describir o mundo con precisión. Esta máquina tiña dificultades para entender, por exemplo, que unha persoa que se afeita cunha máquina de afeitar eléctrica segue sendo unha persoa, xa que consideraba que tiña partes eléctricas no momento do afeitado.

Tras estas dificultades, comprendeuse que os sistemas de intelixencia artificial tiñan que poder adquirir o seu propio coñecemento baseándose en recoñecemento de patróns a través de datos non procesados. De aquí naceu o ***machine learning***.

1.1.1. *Machine learning*

O primeiro fito que se lle atribúe á aprendizaxe automática foi un programa que xogaba ás damas e era capaz de mellorar o seu xogo pola súa conta, e fai preto de 60 anos gañou fronte a Robert Nealey, campión de damas nese momento. Este programa foi deseñado por Arthur Samuel, a quen se lle atribúe o termo "aprendizaxe automática" (Barro, 2022).

O principal interese deste fito non foi a propia vitoria, senón o potencial que demostraba o *software* que tiña detrás xa que, en vez de programar a inmensa cantidade de posibles escenarios que se podían dar no xogo, fixo que reaccionara en función aos xogos pasados. O ordenador "aprende" a xogar ás damas tras múltiples partidas, calculando riscos, sopesando diferentes factores e planificando os seguintes movementos. Estes son os principios nos que se basea o *machine learning* (Bayer, 2019).

Exemplos de algoritmos de *machine learning* son a **regresión loxística**, que pode recomendar cando é necesario facer unha cesárea, ou o algoritmo ***naïve* Bayes** (Webb *et al.*, 2010), que pode separar *email* desexado de *spam*. Pero estes algoritmos dependen fortemente da representación dos datos que se lles provexa. O algoritmo de regresión para recomendar cesáreas precisaba de datos concretos que indicaba o médico (chamados características) pero non podía inferir estes datos ou as decisións finais directamente cunha imaxe dunha resonancia magnética, xa que os píxeis individuais non contiñan a información que precisaba.

É sinxelo resolver algúns problemas de intelixencia artificial escollendo un **conxunto de**

características adecuado e indicándollo a un algoritmo de aprendizaxe automática. Desgraciadamente, para moitas tarefas é complicado determinar que características son necesarias, como por exemplo no caso da detección de coches en fotografías. Poderiamos decidir que é necesario saber se hai rodas na imaxe, pero é complicado definir exactamente como é unha roda en termos de píxeis xa que existen moitos factores que poden alterar a apariencia da mesma (por exemplo, o reflexo do sol, as sombras, ou que parte da vista da roda estea bloqueada por outro obxecto).

1.1.2. *Deep learning*

Unha solución ao problema presentado anteriormente, que constitúe o enfoque coñecido como **aprendizaxe de representación** (*representation learning*), é utilizar aprendizaxe automática para descubrir non só o mapeado da representación á saída, senón tamén a representación en si mesma. Este tipo de aprendizaxe consegue representacións cun funcionamento mellor ao que se pode obter con representacións feitas "a man", e permite aos sistemas de IA adaptarse rapidamente a tarefas con mínima intervención humana.

Un algoritmo de aprendizaxe de representación pode atopar un conxunto de características válido para unha tarefa simple en cuestión de minutos, e de horas ou meses no caso de tarefas complexas, en comparación ás décadas que pode levar de tempo e esforzo humano. Un exemplo clásico é o **autocodificador** (*autoencoder*), que combina unha función de codificación e unha de decodificación e é adestrado para preservar a máxima información posible cando se pasa unha entrada por ambas funcións, á vez que para manter certas propiedades. Podemos ver na Figura 1.1 unha representación do autocodificador.

O deseño de características ou dos algoritmos que as aprenden, xeralmente trata de separar os **factores de variación** que explican os datos observados, que rara vez son cantidades directamente observables. Poden existir como construtos na mente humana que provén explicacións simples e útiles ou causas inferidas dos datos observados. No exemplo do coche mencionado anteriormente, un factor de variación podería ser o ángulo no que ilumina o sol.

A principal dificultade nas aplicacións no mundo real é que a maioría dos factores de variación inflúen en todos os datos que observamos (se é de noite, o coche vaise ver máis escuro). Moitas aplicacións precisan desligar estes factores e descartar aqueles que non nos interesan. É moi complicado extraer características tan abstractas e de tan alto nivel de datos sen procesar. Cando é case igual de difícil obter unha representación que resolver o problema orixinal, a aprendizaxe de representación non parece que axude para estes casos.

A **aprendizaxe profunda** soluciona este problema introducindo representacións que se expresan en termos de outras máis simples. É dicir, permite construír conceptos máis complexos a

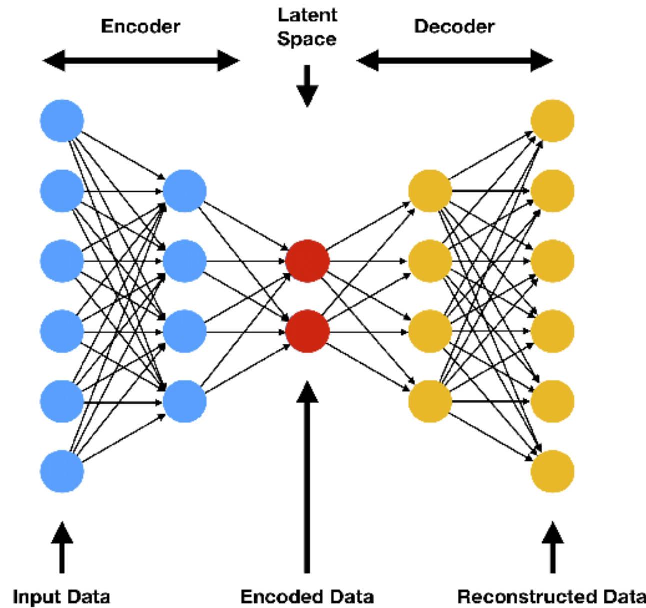


Figura 1.1: Autocodificador (Wang *et al.*, 2021).

partir de conceptos sinxelos.

O exemplo máis típico dun modelo de *deep learning* é a rede de avance profundo ou **perceptrón multicapa** (*multilayer perceptron*, MLP), que se detallará no seguinte capítulo. Un perceptrón multicapa é, en esencia, unha función matemática que transforma certos valores de entrada en valores de saída e está formada por composición de moitas funcións máis simples.

Esta é unha das perspectivas dende as que analizar a aprendizaxe automática, a de obter a representación adecuada para os datos, pero tamén se pode pensar como que a profundidade permite ao ordenador aprender un programa de múltiples pasos. Cada capa da representación pode verse como o estado da memoria do ordenador tras executar outro conxunto de instrucións en paralelo. Redes con máis profundidade poden executar máis instrucións secuencialmente. As instrucións secuenciais teñen moita importancia porque instrucións que se atopan máis adiante poden referirse aos resultados de outras anteriores. Segundo esta visión, non toda a información das activacións dunha capa (concepto que se presentará máis adiante) codifica necesariamente os factores de variación que explican a entrada. A representación tamén garda información de estado que axuda a executar un programa que poida darlle sentido á entrada.

Existen dúas formas de medir a **profundidade** dun modelo: baseándose no **número de instrucións secuenciais** que deben ser executadas para avaliar a arquitectura[1] da rede e baseándose na **profundidade do grafo** que describe como se relacionan os conceptos entre si. A

[1]Unha arquitectura de rede define a forma na que se estrutura un modelo e para que está deseñado.

primeira pode pensarse como a lonxitude do camiño máis longo a través dun diagrama de fluxo que describe como computar cada unha das saídas do modelo dadas as súas entradas. A profundidade cambiará dependendo de que funcións permitimos que se usen como pasos individuais no diagrama (como vemos na Figura 1.2). A segunda é utilizada para os modelos profundos de probabilidade, e a profundidade do diagrama de computacións que se precisa para computar a representación de cada concepto pode ser moito máis profunda que o grafo dos conceptos en si mesmos. Isto débese a que o entendemento do sistema dos conceptos simples pode ser refinado dada información sobre os conceptos máis complexos. Por exemplo, un sistema de IA que analiza unha imaxe na que unha cara ten un ollo en sombra pode recoñecer inicialmente un único ollo pero, tras detectar que hai unha cara, pode inferir que probablemente haxa outro ollo.

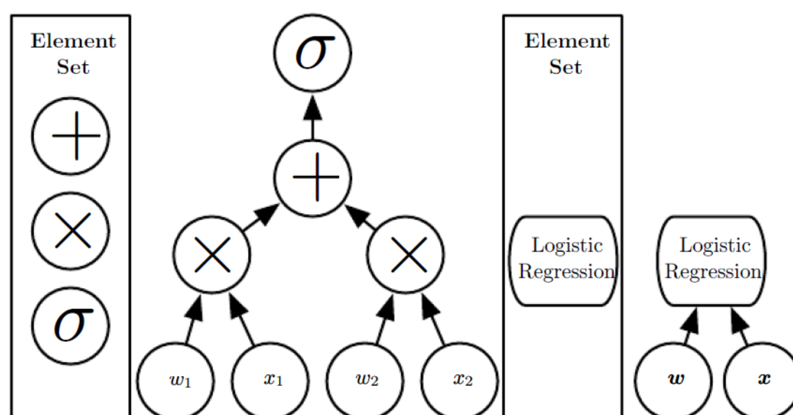


Figura 1.2: Dúas representacións dun mesmo diagrama considerando elementos diferentes (Goodfellow *et al.*, 2016).

Non sempre está claro cal destas dúas formas é máis relevante e, como persoas diferentes escollen diferentes conxuntos dos elementos máis pequenos sobre os que construír os seus grafos, non hai un só valor correcto para a profundidade dunha arquitectura, nin hai un consenso sobre canta profundidade debe ter un modelo para considerarse "profundo". Porén, podemos ver a aprendizaxe profunda como o estudo de modelos que implican unha maior cantidade de composición de funcións aprendidas ou conceptos aprendidos dos que tradicionalmente se utilizan en aprendizaxe automática.

1.1.3. Introdución histórica

En canto ás tendencias ao longo do tempo con respecto á aprendizaxe profunda, destacamos as seguintes:

- Existe dende hai moito tempo con distintos nomes segundo a visión filosófica que se lle

dera, variando a súa popularidade na historia.

- Foi tornando máis útil segundo a cantidade de datos de adestramento aumentou.
- Os modelos de aprendizaxe profunda aumentaron o seu tamaño coa mellora de infraestrutura *software* e *hardware* para *deep learning*.
- Resolve problemas cada vez máis complexos con cada vez máis precisión.

Existiron tres "olas" de desenvolvemento do *deep learning*. Nos seus inicios, entre as décadas dos 40 e 60, coñecíase como **cibernética** (*cybernetics*), a continuación, entre as décadas dos 80 e 90, como **conexionismo** (*connectionism*), e actualmente, dende 2006, como **aprendizaxe profunda** (*deep learning*).

Os primeiros predecesores da aprendizaxe profunda moderna eran modelos lineais simples motivados por unha perspectiva neurocientífica. Esta perspectiva entende que estes modelos están inspirados no cerebro biolóxico, modelando como ocorre ou como podería ocorrer a aprendizaxe no cerebro. Por este motivo, tamén se lles chama redes neuronais artificiais (*Artificial Neural Network*, ANN). Actualmente, o termo moderno *deep learning* vai máis alá desta perspectiva, parecéndose máis ao principio xeral de aprendizaxe de múltiples niveis de composición, e estes modelos poden aplicarse en problemas de aprendizaxe automática que non están necesariamente inspirados pola neurociencia.

Estes primeiros modelos consistían nunha función $f(x, w)$ que recibía unha entrada (un vector x) e uns pesos asociados (un vector w) e obtiña unha saída. A neurona de McCulloch-Pitts foi un primeiro modelo no que un humano tiña que implementar os pesos adecuados para poder resolver o problema correctamente. Na década de 1950, o **perceptrón** (Rosenblatt, 1958, 1960) converteuse no primeiro modelo que podía aprender os pesos que definían unhas certas categorías dados exemplos de entradas de cada categoría. ADALINE (*Adaptive Linear Element*), da mesma época, retornaba simplemente o valor de $f(x)$ en si mesmo para predicir un número real (Widrow & Hoff, 1960), e podía aprender a predicir estes números a partir de datos.

O algoritmo que se utilizaba para adaptar os pesos para o perceptrón é un caso concreto do algoritmo do **descenso de gradiente estocástico** (*stochastic gradient descent*). Variantes deste algoritmo continúan a ser os algoritmos de adestramento dominantes no campo a día de hoxe.

Os modelos baseados nas funcións $f(x, w)$ utilizadas polo perceptrón e ADALINE chámanse **modelos lineais**. Son os máis utilizados en aprendizaxe automática, aínda que actualmente moitos deles son adestrados de maneira diferente aos orixinais. Este tipo de modelos teñen moitas limitacións, como por exemplo que non poden aprender funcións non lineais como a función

XOR[2], o cal causou un descenso na súa popularidade a finais da primeira ola. Non obstante, seguen a ter moita utilidade, e aínda podemos facer algunhas inferencias da neurociencia, conseguindo modelos como o Neocognitron (Fukushima, 1980). Esta é unha poderosa arquitectura de modelos para procesamento de imaxes inspirada pola estrutura do sistema visual dos mamíferos, que máis adiante se converteu na base para as **redes convolucionais** (*convolutional networks*) actuais. A maioría das redes neuronais que temos hoxe en día están baseadas nunha neurona cunha activación de **unidade lineal rectificada** (*rectified linear unit*, ReLU), definida no seguinte capítulo.

A pesar de ser unha boa fonte de inspiración, a neurociencia non debe tomarse como única guía, xa que non temos suficiente coñecemento sobre a aprendizaxe biolóxica como para conseguir os algoritmos que utilizamos para adestrar as arquitecturas de redes neuronais. Actualmente, o *deep learning* consegue inspiración de moitos campos, especialmente de fundamentos de matemáticas como a álxebra lineal, probabilidade, teoría da información e optimización numérica.

A segunda ola da investigación sobre redes neuronais xurdiu do movemento do conexionismo (Banan *et al.*, 2020), ou procesamento distribuído paralelo, no contexto da ciencia cognitiva. A idea central é que un gran número de compoñentes individuais pode obter comportamento intelixente cando se lle dá estrutura de rede. Nesta era xurdiron varios conceptos principais que seguen sendo centrais a día de hoxe. Un deles é a **representación distribuída** (*distributed representation*) de Hinton (2007b), que distribúe as mesmas pezas de información en múltiples capas escalables e independentes. Cada entrada dun sistema debe ser representada por varias características, e cada característica debe estar implicada na representación de varias das entradas posibles.

Outro gran logro deste movemento foi o uso exitoso do algoritmo de **propagación inversa** ou **retropropagación** (*back-propagation*) de Banan *et al.* (2020) e LeCun & Fogelman-Soulié (1987) para adestrar redes neuronais con representacións internas, do que falaremos máis adiante neste traballo.

Na década de 1990 conseguíronse importantes avances modelando secuencias con redes, e Hochreiter & Schmidhuber (1997) introduciron as **redes LSTM** (*Long Short-Term Memory*) para resolver algunhas das dificultades matemáticas fundamentais que se atopaban ao modelar secuencias longas, que engaden unha certa memoria a curto prazo ás neuronas da rede. Hoxe en día utilízanse moito no campo de **procesamento de linguaxe natural** (*natural language processing*, NLP). A terceira ola comezou arredor de 2006, cando Geoffrey Hinton (Hinton, 2007a) demostrou que un tipo de rede chamado “rede de crenza profunda” (*deep belief network*) podía

[2] A función XOR recibe un vector $x = (x_1, x_2)$, onde x_1 e x_2 só poden ter valores 0 ou 1, e as súas posibles saídas son $f([0,1], w) = 1$, $f([1,0], w) = 1$, $f([1,1], w) = 0$ e $f([0,0], w) = 0$. Esta función representa a desigualdade, é dicir, que dúas entradas sexan diferentes, e é moi utilizada en informática.

ser adestrada eficientemente utilizando unha estratexia chamada pre-adestramento voraz por capas (*greedy layer-wise pre-training*). Pronto se descubriu que esa estratexia podía utilizarse para adestrar moitos outros tipos de redes profundas e que axudaba sistematicamente a mellorar a xeneralización en exemplos de proba. A xeneralización nun modelo é a súa capacidade de obter resultados adecuados para conxuntos de datos non observados previamente.

Existen dous tipos principais de aprendizaxe en *machine learning*: a **aprendizaxe supervisada** e a **aprendizaxe non supervisada**. Ambas consisten en observar varios exemplos dun vector aleatorio x pero a diferenza principal é que a aprendizaxe non supervisada pretende descubrir, implícita ou explicitamente, a súa distribución de probabilidade $p(x)$, ou propiedades interesantes da mesma, mentres que a aprendizaxe supervisada observa tamén un valor ou vector asociado y e aprende a predicir y a partir de x, xeralmente estimando a probabilidade dependente $p(y|x)$.

Esta era comezou cun enfoque en novas técnicas de aprendizaxe non supervisada e na habilidade de modelos profundos de xeneralizar ben a partir de conxuntos de datos pequenos, aínda que hoxe hai más interese en algoritmos supervisados máis antigos e na habilidade dos modelos profundos de aproveitar grandes conxuntos de datos etiquetados.

Dende 2016 considérase unha regra aproximada que un algoritmo supervisado de aprendizaxe profunda pode conseguir un rendemento aceptable a partir duns 5000 exemplos etiquetados por categoría, e obterá rendemento polo menos igual que aquel dun humano cando se adestre cun conxunto de datos que conteña un mínimo de 10 millóns de exemplos etiquetados.

A partir da aparición de unidades ocultas, as redes neuronais artificiais duplicaron o seu tamaño preto de cada 2.4 anos, grazas aos avances en tecnoloxía que permiten ter ordenadores máis rápidos con maior memoria e á dispoñibilidade de conxuntos de datos máis grandes.

1.2. Coñecementos previos

Deben terse certos coñecementos previos das áreas de álxebra lineal, probabilidade e optimización numérica para entender certos algoritmos utilizados en *deep learning*, que se poden atopar resumidos nos capítulos 2 a 4 no libro de Goodfellow *et al.* (2016). Tamén é recomendable ter coñecementos previos sobre *machine learning*, atopados no capítulo 5 do mesmo libro. Outros recursos interesantes son as publicacións de Russell & Norvig (2016) e Zhang *et al.* (2021).

As tarefas de aprendizaxe automática son descritas habitualmente en termos de como o sistema debe procesar un exemplo. Defínese un exemplo como unha colección de características medidas cuantitativamente sobre un obxecto ou evento que queremos que o sistema de aprendizaxe automática procese. Denótase un exemplo por un vector $x \in \mathbb{R}^n$, onde cada elemento x_i,

$i \in \{1, ..., n\}$, $n \in \mathbb{N}$, representa unha característica distinta e $x_i = 0$ representa a ausencia da característica i.

Existen moitos tipos diferentes de tarefas de *machine learning*, entre as que se atopan:

- **Clasificación**: Consiste en clasificar a entrada nunha categoría. Pode representarse por $f(x) = y$, $y \in \{1, \ldots, k\}$ para k categorías. Outras representacións de y son a dun vector composto por 0 salvo o elemento correspondente á categoría i (que será un 1) e a dun vector de compoñentes non negativas, que suman 1 e representan a probabilidade de que a entrada pertenza a cada categoría.
- **Regresión**: Trata de predicir un valor numérico dada unha entrada, é dicir, $f(x) = y$, $y \in \mathbb{R}$.
- **Detección de anomalías**: Consiste en identificar obxectos ou eventos inusuais ou atípicos nun conxunto dado.
- **Tradución automática**: Consiste en traducir unha secuencia de símbolos dun linguaxe a outro (non só para linguaxe natural).

Para avaliar os modelos de *machine learning* pódense empregar diferentes métricas (Powers, 2020). Ademais das funcións de custo e perda, que se utilizan para optimizar os modelos, existen moitas métricas de avaliación entre as que se atopan a precisión (*precision*) e a exhaustividade (*recall*), tamén coñecida como sensibilidade (*sensitivity*). Estas métricas non se empregan para adestrar ao modelo e non teñen por que ser diferenciables.

1.2.1. Sobreaxuste e subaxuste

Os algoritmos de *machine learning* deben realizar un adestramento para aprender a axustar os parámetros que utilizan para realizar as súas respectivas tarefas. Para poder adestrar os modelos e comprobar a súa eficacia deben dividirse os conxuntos de datos de exemplo en diferentes grupos, cada un cun obxectivo distinto. Estes subconxuntos son as particións de adestramento (*training*), validación (*validation*) e probas (*test*).

Os datos da partición de **adestramento** empréganse para adestrar ao modelo e aprender a partir deles como axustar os parámetros necesarios. Por outro lado, a partición de **validación** serve para validar a evolución do adestramento do modelo, avaliando o seu rendemento para decidir se é necesario continuar o adestramento, se debe modificarse ou se se pode finalizar. Por último, a partición de **probas** utilízase unha vez o modelo finalizou de adestrar para estimar o seu comportamento con datos novos e avaliar así o seu rendemento real.

A división dos datos de exemplo nestes tres subconxuntos, cando se fai nunha proporción adecuada, asegura que o modelo sexa capaz de xeneralizar os resultados para datos non observados previamente. Se a partición de adestramento é demasiado grande, córrese o risco de incorrer nun **sobreaxuste** (*overfitting*), é dicir, que o modelo sexa capaz de reproducir á perfección os datos sobre os que foi adestrado pero xera resultados artificiais e comete moitos erros cando se lle presentan novos datos. Matemáticamente, isto ocorre cando a diferencia entre o erro de adestramento e o erro de probas é demasiado grande. Pola contra, cando as particións de probas e validación son demasiado grandes aparece o risco dun **subaxuste** (*underfitting*). Ao contrario que o sobreaxuste, un subaxuste ocorre cando o modelo non é capaz de realizar boas predicións sobre os datos de adestramento, pero tampouco sobre datos novos. Isto sucede se o modelo non é capaz de obter un erro de adestramento o suficientemente pequeno. Podemos observar na Figura 1.3 unha comparativa das gráficas dun algoritmo sobreaxustado, subaxustado e equilibrado.

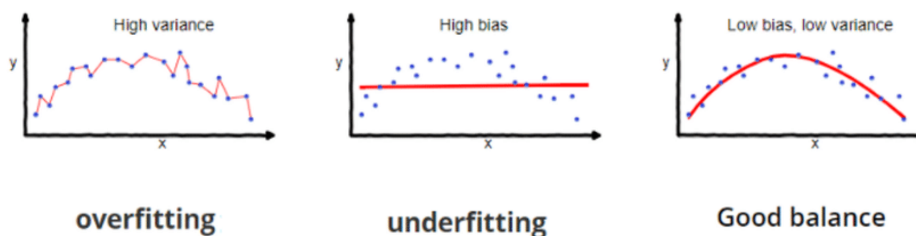


Figura 1.3: Comparación das gráficas dun algoritmo sobreaxustado, subaxustado e equilibrado (Lucas, 2019).

Capítulo 2

Modelos e algoritmos

Neste capítulo presentaranse os conceptos básicos necesarios para comprender as redes neuronais. Comezarase definindo unha rede neuronal e varias funcións de activación. A continuación, presentaranse diferentes tipos de redes neuronais, funcións de custo e perda e algoritmos de aprendizaxe e, finalmente, explicarase o concepto de hiperparámetros e algúns dos seus algoritmos de optimización.

2.1. Funcións de activación

As redes neuronais están formadas por capas de nodos (tamén chamados neuronas ou unidades), nas que sempre hai unha capa de entrada, unha de saída e, polo menos, unha capa oculta (salvo no caso do perceptrón simple do que falaremos nesta sección). Cada neurona conéctase a outra cun peso asociado (IBM, 2020a). Cando se determina unha capa de entrada, decídense unhas ponderacións asociadas a cada elemento do vector de entrada, que indican a influencia de cada un deles na saída. Multiplícanse os elementos da entrada polas súas ponderacións ou pesos e súmanse, para aplicarlle unha **función de activación**. A operación que se realiza previamente á función de activación denomínase **función de entrada**, definida formalmente como segue:

Definición 2.1. Unha función de entrada, in_j, é aquela que, a cada vector de entrada (*input*) $(a_1, \dots, a_n) \in \mathbb{R}^n$, con $n \in \mathbb{N}$, dunha unidade $j \in \mathbb{N}$, dende as unidades $i \in \{1, \dots, n\}$, respectivamente, asigna o valor $z \in \mathbb{R}$ dado por:

$$in_j(a_1, \dots, a_n) = \sum_{i=0}^{n} w_{i,j} a_i = z,$$

onde $w_{i,j} \in \mathbb{R}$ son os pesos asociados a cada un dos enlaces dende i ata j con $i \in \{1, \dots, n\}$. Ademais, de modo análogo á regresión lineal, cada unidade j ten unha entrada *dummy* $a_0 = 1$

con peso asociado $w_{0,j}$ denominado sesgo[1].

Na Figura 2.1 obsérvase unha representación dunha función de activación. Descrito en termos matemáticos, teríamos a seguinte definición de función de activación (Russell & Norvig, 2016).

Definición 2.2. Unha función de activación, $f : \mathbb{R} \to \mathbb{R}$ é aquela que a cada $z \in \mathbb{R}$ asígnalle un valor $a_j \in \mathbb{R}$. Isto é,

$$f(z) = f\left(in_j(x)\right) = a_j,$$

onde $x = (a_1, \ldots, a_n)$, $n \in \mathbb{N}$ é o vector de entrada da unidade j, $j \in \mathbb{N}$. É dicir, f é unha función que converte nunha saída (*output*) dunha unidade a suma ponderada dos elementos de entrada dende un conxunto de unidades enlazadas.

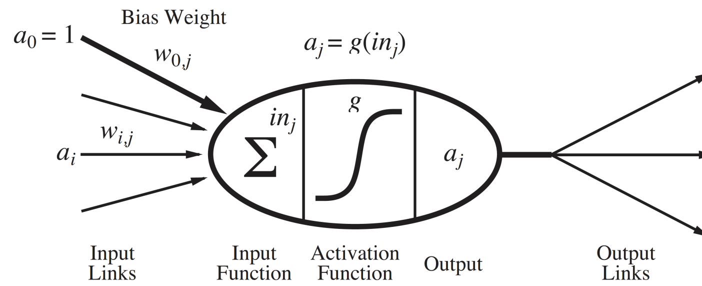


Figura 2.1: Representación dunha función de activación (Russell & Norvig, 2016).

A función de activación é habitualmente un limiar estrito, en cuxo caso a unidade se chama **perceptrón**, ou unha función loxística, en cuxo caso a chamamos **perceptrón sigmoide**. O limiar, ou sesgo, indica cando unha neurona debe activarse e pasar a súa saída á seguinte capa e, se o valor de saída non supera o limiar, os datos non se pasan á seguinte capa.

Definición 2.3. Denomínase función de activación dun perceptrón a función de activación dada por:

$$f(z) = \begin{cases} 1 & \text{se} \quad z \geq 0, \\ \\ 0 & \text{se} \quad z < 0. \end{cases}$$

Definición 2.4. Denomínase función de activación dun perceptrón sigmoide a función de activación dada pola función sigmoide:

$$f(z) = \sigma(z) = \frac{1}{1 + e^{-z}}.$$

Nótese que esta función é diferenciable.

[1] Nalgunhas representacións $z = \sum_{i=1}^{n} w_{i,j} a_i + b$, onde $b \in \mathbb{R}$ é o sesgo.

Ademais destas dúas funcións de activación, existen moitas outras (Rasamoelina *et al.*, 2020), a maioría delas non lineais, dado que o uso exclusivo de funcións lineais implicaría que a saída só podería ser unha función lineal (a composición de funcións lineais é lineal). Algúns modelos combinan capas con funcións lineais e non lineais para poder aproximar funcións non lineais. Algunhas das funcións de activación máis utilizadas son a función tanxente hiperbólica (tanh), porque dá mellores resultados que a sigmoide para redes multicapa, a función ReLU (*Rectified Linear Unit*) e as súas variantes, que trata de forma efectiva o problema do desvanecemento de gradiente, do que falaremos posteriormente, e a función SELU, que induce propiedades de autonormalización. Estas tres funcións están definidas como segue.

Definición 2.5. Denomínase función de activación tanxente hiperbólica a función de activación dada por:

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

Definición 2.6. Denomínase función de activación lineal rectificada a función de activación dada por:

$$f(z) = \mathrm{ReLU}(z) = \text{máx}\{0, z\}.$$

Definición 2.7. Denomínase función de activación **SELU** a función de activación dada por:

$$f(z) = \mathrm{SELU}(z) = \begin{cases} \lambda z & \text{se} \quad z \geq 0, \\ \\ \lambda\alpha(e^z - 1) & \text{se} \quad z < 0, \end{cases}$$

onde $\lambda \approx 1,0507$ e $\alpha \approx 1,6733$.

Existe tamén outro tipo de funcións, coñecidas como funcións de saída[2], que se utilizan nas últimas capas das redes para transformar a saída da capa anterior á forma desexada para o uso dunha función de custo ou perda, das que falaremos nun apartado posterior. As máis coñecidas son as funcións **softmax** e **argmax**, definidas a continuación. A primeira transforma un vector de entrada nun vector de probabilidades, mentres que a segunda o transforma nun vector onde todos os valores son 0 salvo o máximo, que é 1.

Definición 2.8. Dado un vector de entrada $z \in \mathbb{R}^K$, $K \in \mathbb{N}$, denomínase función softmax: $\mathbb{R}^K \to \mathbb{R}^K$, ou función exponencial normalizada, a función de saída dada por:

$$\mathrm{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}},$$

con $i \in \{1, \dots, K\}$. Nótese que $\mathrm{softmax}(z)_i \in (0, 1)$ e $\sum_{i=1}^{K} \mathrm{softmax}(z)_i = 1$.

[2]Na literatura denomínanse habitualmente funcións de activación, a pesar de que esta denominación é errónea ao non coincidir os seus dominios.

Definición 2.9. Dado un vector de entrada $z \in \mathbb{R}^K$, $K \in \mathbb{N}$, denomínase función argmax: $\mathbb{R}^K \to \mathbb{R}^K$ a función de saída dada por:

$$\text{argmax}(z)_i = \begin{cases} 1 & \text{se} \quad z_i = \text{máx}\{z_j \mid j = 1, \dots, K\}, \\ \\ 0 & \text{noutro caso,} \end{cases}$$

onde $i \in \{1, \dots, K\}$.

Dependendo da implementación, se existen varios máximos, a función argmax retorna 1 para todos eles, soamente para o primeiro, ou $\frac{1}{Nmax}$, con $Nmax$ o número de máximos, para cada un deles.

Para determinar os pesos recórrese ao adestramento no que, dados uns conxuntos de datos de adestramento e validación, as ponderacións axústanse segundo a importancia de cada unha das variables de entrada, de modo que as que teñen un valor máis grande contribúen máis ao valor da saída que as outras. Para isto utilízanse diferentes algoritmos dos que falaremos nunha sección posterior.

2.2. Tipos de redes

Poden existir tantas redes como problemas se nos ocorran, pero nesta sección describiremos os tipos máis coñecidos de redes neuronais: o perceptrón, as redes de propagación cara diante, as redes recorrentes e as convolucionais. A maioría de redes mesturan capas de diferentes tipos para obter os resultados desexados.

2.2.1. Perceptrón

O **perceptrón** é a forma máis simple dunha rede neuronal, formada por unha soa neurona que recibe todas as entradas e obtén unha saída. Rosenblatt (1958) foi quen desenvolveu esta neurona, baseándose no traballo de McCulloch & Pitts (1943) e engadíndolle os pesos que se utilizan hoxe en día.

2.2.2. Redes de propagación cara diante

A continuación do perceptrón, as **redes de propagación cara diante** (*feedforward networks*) son as máis básicas. Están formadas por varias capas de neuronas (en paralelo) conectadas entre si nunha soa dirección (en serie). Este tipo de rede non ten ningún estado interno e, polo tanto, poderíamos dicir que non ten “memoria”.

As neuronas de cada capa reciben información soamente da capa inmediatamente anterior. Hai dous tipos principais: redes de capa única e redes multicapa. Nas primeiras, cada unidade conecta directamente as entradas da rede coas súas saídas. Nas segundas teñen unha ou máis capas de **unidades ocultas** que non están conectadas ás saídas da rede. Este tipo é o máis habitual, polo que case sempre se lles chama **perceptrón multicapa** ás redes de propagación cara diante (Figura 2.2). A pesar deste nome, poden estar formadas por neuronas sigmoides (e habitualmente o están para tratar problemas non lineais).

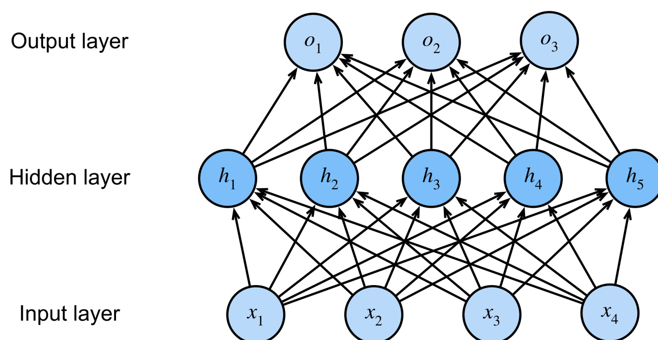


Figura 2.2: Representación dun perceptrón multicapa (Zhang *et al.*, 2021).

As redes formadas por unha capa de perceptróns son moi útiles para casos de clasificación onde os resultados son separables linealmente, pero teñen dificultades para resolver problemas non lineais, como por exemplo a función XOR explicada no primeiro capítulo.

2.2.3. Redes recorrentes

A diferencia das redes de propagación cara diante, as **redes recorrentes** (*recurrent neural networks*, RNN) retornan a súa saída como parte da súa propia entrada, é dicir, teñen **retroalimentación**. Este valor é denominado **estado oculto** (*hidden state*). Isto quere dicir que a resposta da rede a unha entrada concreta depende do seu estado inicial, que pode depender de entradas anteriores. Considérase que este tipo de redes, a diferencia das redes de propagación cara diante, teñen memoria a curto prazo (Russell & Norvig, 2016).

Están baseadas no traballo de Rumelhart *et al.* (1986), onde se introduciu o concepto de *backpropagation*, ou **retropropagación**, que describiremos en detalle na sección de algoritmos. Este algoritmo (ou máis ben unha variación) é o que utiliza a rede para actualizar os seus pesos, xunto co descenso de gradiente.

As redes recorrentes utilízanse principalmente para procesamento de secuencias (Graves, 2014), como predicións a partir de datos de series temporais no campo da minería de proce-

sos, ou no procesamento de linguaxe natural (IBM, 2020b). Para adaptarse a esta arquitectura, desenvolveuse unha variación do algoritmo de retropropagación denominada **retropropagación a través do tempo** (*backpropagation through time*). Estas redes teñen dous tipos de representacións visuais (Figura 2.3), que chamaremos compacta e desenrolada.

A **representación compacta**, que podemos observar no lado esquerdo da Figura 2.3, é a representación da arquitectura da rede, incluíndo todas as súas capas e o fluxo da información ilustrado por frechas. O vector de entrada está representado por x, o de saída por y e o estado oculto por h.

A **representación desenrolada** representa as capas individuais, ou pasos temporais, da rede como se fora unha rede de avance cara diante. Na Figura 2.3, a $t \in \mathbb{N}$ indica o **paso temporal**, é dicir, o momento no que entra unha entrada concreta á rede, xerando a saída correspondente, x^t representan as entradas da rede no paso temporal t, y^t as saídas, h^t os estados ocultos e as frechas entre as capas ocultas representan a retroalimentación da información da rede. As ecuacións correspondentes ao estado oculto h^t e á saída y^t en cada momento t veñen dadas por:

$$h^t = f(b + Wh^{t-1} + Ux^t),$$

$$y^t = c + Vh^t,$$

onde b e c representan os respectivos sesgos e U, V e W os pesos asociados a cada unha das variables. Habitualmente a función de activación f empregada é a tanxente hiperbólica.

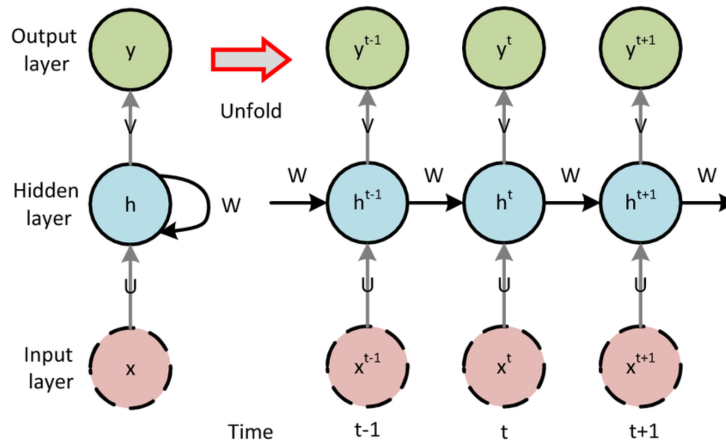


Figura 2.3: Representación compacta e desenrolada dunha rede recorrente (Zakeri Nasrabadi *et al.*, 2021).

O número de capas a desenrolar neste tipo de representación é dependente da secuencia a analizar. Canta máis lonxitude temporal teña a secuencia, maior será o número de capas a representar. Se a secuencia é moi longa, pode aparecer un desvanecemento de gradiente (*vanishing*

gradient), o que implica que a rede vai esquecendo os valores máis antigos da súa retroalimentación. Para solucionar isto, en lugar de utilizar unha rede recorrente simple como a explicada ata agora, utilízanse unhas unidades chamadas celas **LSTM** ou celas **GRU** (*Gated Recurrent Unit*) que alivian o problema do desvanecemento de gradiente ao permitir que este flúa a través da cela, modificando o valor do estado oculto só cando sexa necesario.

LSTM

Cando hai moito espazo entre a saída que queremos predicir e a información que dá o contexto necesario, as RNNs non teñen a suficiente memoria. Por exemplo, tomemos o texto "Crecín en Francia [...] Falo moi ben *francés*", onde o inicio e o final están separados por varias frases en medio. Para poder predicir a palabra "*francés*" que continúa a frase, necesitaríamos coñecer o contexto do inicio, que nos indica que creceu en Francia (Olah, 2015). Por esta razón Hochreiter & Schmidhuber (1997) introduciron por primeira vez as redes LSTM. As celas LSTM reciben unha entrada, a retroalimentación do paso temporal anterior (o estado oculto) e un estado da cela (no exemplo anterior, o estado da cela contería a información do inicio da frase, que creceu en Francia), que modifican en base aos outros dous elementos. As LSTM deciden que información eliminar do estado (a información irrelevante, como as frases que poidan estar no medio no exemplo), que información nova engadirlle (que fala moi ben un idioma) e, finalmente, que información do estado devolven como saída (neste caso, a palabra "francés").

As RNN convencionais teñen unha cadea de capas que se repiten cunha estrutura moi simple, como unha capa de neuronas con función de activación tanh. As redes LSTM, en cambio, están formadas por capas de celas LSTM, compostas por varias estruturas (tres para ser exactos) que interactúan como se amosa na Figura 2.4. Manteñen o estado oculto (ou entrada recorrente) h_t que introducimos nas redes recorrentes pero, ademais, engaden un estado da cela C_t que contén máis información que o estado oculto. Este novo estado é polo que se di que as redes LSTM teñen máis memoria, xa que almacena información sobre saídas anteriores, mantendo a que se considera importante e eliminando a que non.

A liña superior na figura indica o estado da cela no paso temporal anterior C_{t-1}, $t \in \mathbb{N}$, e actúa como unha cinta transportadora, cunhas poucas interaccións lineais que modifican o seu valor ata a saída (Olah, 2015). A cela LSTM ten a capacidade de engadir ou eliminar información do estado mediante estruturas chamadas **portas**. As portas, representadas na figura por f_t, i_t e o_t, están formadas por unha función sigmoide e unha operación de produto Hadamard (Definición 2.10), representado na figura por $\times$, e son unha forma de deixar pasar información opcionalmente. Canta información pasa vén dado pola saída da función sigmoide, representada na imaxe por σ, cun 0 indicando que non pase nada e canto máis preto dun 1 está máis información deixa pasar.

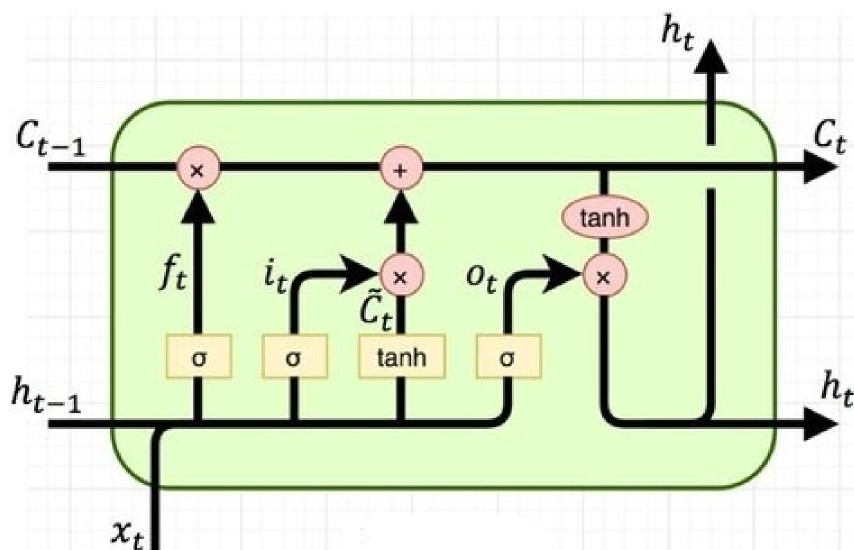


Figura 2.4: Representación dunha cela LSTM (Varsamopoulos *et al.*, 2018).

Definición 2.10. Sexan A, $B \in \mathbb{R}^{n\times m}$, $n, m \in \mathbb{N}$, $a_{ij} \in A$, $b_{ij} \in B$, $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, m\}$. Defínese a multiplicación elemento a elemento como segue:

$$A \circ B = C = (a_{ij} \times b_{ij}),$$

onde $C \in \mathbb{R}^{n\times m}$. Denomínase esta multiplicación como produto Hadamard ou produto de Schur.

Unha cela LSTM ten tres destas portas para controlar o estado da cela. Como vemos na Figura 2.4, a primeira delas, f_t, decide que información das anteriores saídas continúa. A estrutura que toma a decisión denomínase a **porta de esquecemento** (*forget gate*). En función da entrada nese paso temporal x_t, $t \in \mathbb{N}$, e o estado oculto da rede h_{t-1}, obtense un vector composto por valores entre 0 e 1 empregando a función sigmoide que se combina con cada elemento do estado anterior da cela C_{t-1} mediante o produto Hadamard, indicando a información que manter.

A continuación decídese a información que engadir ao estado en dous pasos. Primeiro, a **porta de entrada** (*input gate*) i_t decide que valores do estado se deben actualizar xerando un vector composto por números entre 0 e 1 cunha función sigmoide e despois a capa tanh crea un vector cos novos valores candidatos $\tilde{C}_t$ que poden engadirse ao estado. Os vectores obtidos combínanse grazas ao produto Hadamard e engádese a nova información ao estado da cela mediante unha suma de vectores.

Por último, decidimos a saída que queremos xerar, que será unha versión filtrada do estado da cela. Por un lado, a **porta de saída** (*output gate*) o_t decide que información do estado queremos devolver. Por outro lado, pasamos o estado da cela pola función tanh para converter os seus valores ao rango $[-1, 1]$. Ambos vectores se combinan mediante un produto Hadamard para xerar a saída que devolve a cela, que será tamén o estado oculto h_t.

Desta cela obtemos por un lado a saída e estado oculto h_t, que será a entrada recorrente do seguinte paso temporal da rede, e o novo estado da cela C_t, que será o estado inicial no seguinte paso. Formalmente, as ecuacións correspondentes a cada unha das portas f_t, i_t e o_t son:

$$f_t = \sigma\left(U_{f_t} x_t + W_{f_t} h_{t-1}\right),$$

$$i_t = \sigma\left(U_{i_t} x_t + W_{i_t} h_{t-1}\right),$$

$$o_t = \sigma\left(U_{o_t} x_t + W_{o_t} h_{t-1}\right),$$

sendo U e W os pesos das variables de entrada e do estado oculto para cada respectiva porta. Por outro lado, as ecuacións dos estados da cela, onde $\tilde{C}_t$ é o estado actual e C_t o estado de saída, e a súa saída h_t son:

$$\tilde{C}_t = \tanh\left(U_{\tilde{C}_t} x_t + W_{\tilde{C}_t} h_{t-1}\right),$$

$$C_t = f_t \circ C_{t-1} + i_t \circ \tilde{C}_t,$$

$$h_t = \tanh\left(C_t\right) \circ o_t,$$

onde $U_{\tilde{C}_t}$ e $W_{\tilde{C}_t}$ son os pesos das variables de entrada e do estado oculto para o estado actual.

Existen outras variantes das LSTM, pero esta é a máis común. Outra moi utilizada e moi popular por ser máis simple é a GRU. En Greff *et al.* (2016) pode verse unha comparación dos resultados de diferentes variantes, concluíndo en que son bastante similares.

GRU

As redes GRU foron introducidas por Cho *et al.* (2014). Para resolver o problema do desvanecemento do gradiente dunha RNN convencional, GRU utiliza unha porta de actualización (*update gate*), combinación das portas de entrada e esquecemento dunha LSTM, e unha porta de reinicio (*reset gate*). Ademais, une o estado da cela co estado oculto (a entrada recorrente) e fai outros pequenos cambios que simplifican a cela LSTM, como se pode observar na Figura 2.5 (Olah, 2015).

As dúas portas deciden que información se lle pasa á saída. Poden adestrarse para manter información moi antiga, sen que se desvaneza no tempo, ou eliminar información que é irrelevante para a predición (Kostadinov, 2017).

Primeiro calculamos o resultado da porta de actualización z_t para o instante t, coa seguinte fórmula:

$$z_t = \sigma(W^{(z)} x_t + U^{(z)} h_{t-1}),$$

onde $W^{(z)}$ e $U^{(z)}$ son, respectivamente, os pesos de x_t e h_{t-1}, que almacena a información dos $t-1$ pasos anteriores. Súmanse ambas variables multiplicadas polo seu peso e aplícase unha función

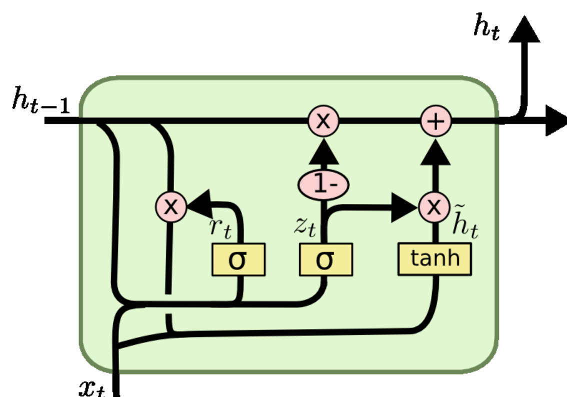


Figura 2.5: Representación dunha cela GRU (Abdulwahab *et al.*, 2017).

sigmoide para representalo entre 0 e 1. Isto decide canta información mantemos das anteriores iteracións.

A continuación, calculamos o resultado da porta de reinicio r_t para o instante t coa fórmula:

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}).$$

A fórmula é equivalente á da porta de actualización pero os pesos son diferentes, así como o uso que se lle dá, xa que esta decide canta información esquecer.

O resultado desta porta combínase cunha función tanh:

$$\tilde{h}_t = \tanh(W^{(\tilde{h})}x_t + r_t \circ U^{(\tilde{h})}h_{t-1}).$$

De novo, estes pesos son diferentes aos anteriores. Finalmente calculamos h_t, que será a saída para o instante t que se lle pasará ás futuras iteracións, en combinación co resultado da porta de actualización:

$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t.$$

2.2.4. Redes convolucionais

As redes convolucionais (*convolutional neural networks*, CNN) son moi similares ás redes de propagación cara diante, coa diferencia de que se aproveitan maioritariamente dos principios da álxebra lineal, como a multiplicación de matrices, para recoñecemento de patróns en imaxes (IBM, 2020a). Utilízanse tamén para tarefas de clasificación e tarefas de visión por ordenador (*computer vision tasks*), un campo da intelixencia artificial que permite a ordenadores e sistemas obter información significativa de imaxes dixitais, vídeos e máis, e utilizar esa información para, por exemplo, dar suxerencias de xente que pode saír nunha fotografía para etiquetalos nunha rede social. Podemos ver un exemplo na publicación de Simonyan & Zisserman (2015).

Atribúese o seu descubrimento a LeCun *et al.* (1989), coa base no traballo de Fukushima (1980). Son bastante esixentes computacionalmente e requiren habitualmente tarxetas gráficas (*graphics processing units*, GPU) para adestrar aos modelos. Son equivariantes (Definición 2.12) a traslacións, pero non invariantes (Definición 2.11). É dicir, a saída dun elemento trasladado estará tamén trasladada.

Definición 2.11. Sexa $g : \mathbb{R}^n \to \mathbb{R}^n$, $n \in \mathbb{N}$, unha función de traslación arbitraria. Dicimos que unha función $f : \mathbb{R}^n \to \mathbb{R}^m$, $m \in \mathbb{N}$, é invariante a traslacións en $I \subset \mathbb{R}^n$ se temos:

$$f(g(I)) = f(I).$$

Definición 2.12. Sexa $g : \mathbb{R}^n \to \mathbb{R}^n$, $n \in \mathbb{N}$, unha función de traslación arbitraria. Dicimos que unha función $f : \mathbb{R}^n \to \mathbb{R}^n$ é equivariante a traslacións en $I \subset \mathbb{R}^n$ se existe unha función $g' : \mathbb{R}^n \to \mathbb{R}^n$ que reflicta a traslación dunha forma significante de modo que:

$$f(g(I)) = g'(f(I)).$$

Utilizan principalmente tres tipos de capas:

1. Capas convolucionais.
2. Capas de *pooling*.
3. Capas completamente conectadas (*fully-connected layers*, FCL).

A primeira capa da rede é convolucional. Pode estar seguida doutras capas convolucionais ou de *pooling*, e a última capa é unha FCL. Con cada capa a CNN aumenta a súa complexidade para identificar partes máis grandes da imaxe (IBM, 2020c). As capas iniciais identifican características simples como cores e bordes e segundo a información vai pasando por máis capas da rede empezan a recoñecerse formas e outros elementos grandes ata que se identifica o obxecto que se pretendía atopar.

Capa convolucional

A capa convolucional precisa de tres elementos: información de entrada (*input data*), un filtro e un mapa de características (*feature map*). Se a entrada é unha imaxe con cor en formato RGB[3], a información de entrada terá tres dimensións: altura, anchura e profundidade. O kernel ou filtro, tamén chamado detector de características, desprázase a través dos campos receptivos (a parte

[3]O modelo RGB representa cada píxel dunha imaxe por tres compoñentes correspondentes ás cores primarias: vermello, verde e azul.

da matriz á que aplicamos o filtro en cada paso) da imaxe comprobando se unha característica concreta está presente, o que se coñece como convolución.

O detector de características é unha matriz de pesos que representa parte da imaxe. O tamaño do filtro pode variar, aínda que usualmente é unha matriz 3×3, e isto determina tamén o tamaño do campo receptivo. Tras decidir o tamaño, o filtro aplícase a unha área da imaxe, calculando o produto Hadamard dos píxeis de entrada e o filtro. Este produto introdúcese nunha matriz de saída. A continuación, móvese o filtro e o proceso repítese ata que se teña procesada toda a imaxe. A saída final desta serie de produtos Hadamard entre a entrada e filtro chámase mapa de características, mapa de activación ou característica convolucionada. Vexamos como se desenvolve este proceso no exemplo a continuación.

Exemplo 2.13. Sexa $A \in \mathbb{R}^{4\times 5}$, definida como segue, a matriz de entrada dunha capa convolucional nunha rede convolucional

$$A = \begin{pmatrix} 1 & 0 & 2 & 4 & 1 \\ 0 & 2 & 3 & 0 & 1 \\ 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix},$$

e sexa $F \in \mathbb{R}^{3\times 3}$ o filtro desa mesma capa, definido da seguinte forma:

$$F = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 3 & 3 \\ 2 & 1 & 0 \end{pmatrix}.$$

Se aplicamos o filtro á matriz empezando polo elemento $a_{1,1}$, obtemos a saída:

$$v_{1,1} = (1\cdot 1) + (1\cdot 0) + (0\cdot 2) + (1\cdot 0) + (3\cdot 2) + (3\cdot 3) + (2\cdot 1) + (1\cdot 3) + (0\cdot 0) = 21.$$

Aplicamos agora o filtro movéndoo unha columna á dereita, empezando polo elemento $a_{1,2}$, obtendo:

$$v_{1,2} = (1\cdot 0) + (1\cdot 2) + (0\cdot 4) + (1\cdot 2) + (3\cdot 3) + (3\cdot 0) + (2\cdot 3) + (1\cdot 0) + (0\cdot 0) = 19.$$

Movemos outra vez o filtro á dereita, empezando polo elemento $a_{1,3}$:

$$v_{1,3} = (1\cdot 2) + (1\cdot 4) + (0\cdot 1) + (1\cdot 3) + (3\cdot 0) + (3\cdot 1) + (2\cdot 0) + (1\cdot 0) + (0\cdot 0) = 12.$$

Como xa non podemos movelo máis á dereita, aplicamos agora o filtro empezando polo elemento $a_{2,1}$:

$$v_{2,1} = (1\cdot 0) + (1\cdot 2) + (0\cdot 3) + (1\cdot 1) + (3\cdot 3) + (3\cdot 0) + (2\cdot 0) + (1\cdot 0) + (0\cdot 1) = 12.$$

Calculamos o seguinte elemento da matriz v comezando polo elemento $a_{2,2}$:

$$v_{2,2} = (1 \cdot 2) + (1 \cdot 3) + (0 \cdot 0) + (1 \cdot 3) + (3 \cdot 0) + (3 \cdot 0) + (2 \cdot 0) + (1 \cdot 1) + (0 \cdot 1) = 9.$$

Por último, calculamos o último elemento comezando polo elemento $a_{2,3}$:

$$v_{2,3} = (1 \cdot 3) + (1 \cdot 0) + (0 \cdot 1) + (1 \cdot 0) + (3 \cdot 0) + (3 \cdot 0) + (2 \cdot 1) + (1 \cdot 1) + (0 \cdot 0) = 6.$$

Observamos o seguinte patrón: os índices do elemento que calculamos da matriz v de saída correspóndense cos índices do elemento da matriz de entrada polo que comezamos a aplicar o filtro en cada paso. Finalmente temos a matriz:

$$v = \begin{pmatrix} 21 & 19 & 12 \\ 12 & 9 & 6 \end{pmatrix}.$$

No primeiro paso, o campo receptivo está formado polos elementos $a_{1,1}$, $a_{1,2}$, $a_{1,3}$, $a_{2,1}$, $a_{2,2}$, $a_{2,3}$, $a_{3,1}$, $a_{3,2}$, $a_{3,3}$:

$$campo_receptivo = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 2 & 3 \\ 1 & 3 & 0 \end{pmatrix}.$$

Podemos tamén deducir deste exemplo que o tamaño da matriz de saída depende do tamaño da matriz de entrada e do filtro. Máis especificamente, vemos que o seu tamaño é de $(4-3+1) \times (5-3+1) = 2 \times 3$. Xeneralizando, se a matriz de entrada A ten tamaño $n_1 \times n_2$ e o filtro F ten tamaño $m_1 \times m_2$, a matriz de saída v terá tamaño $(n_1 - m_1 + 1) \times (n_2 - m_2 + 1)$. □

No caso de que a matriz de entrada teña un tamaño menor ao do filtro nalgunha das súas dimensións, utilízase *zero-padding*, é dicir, recheo con ceros. Esta técnica consiste en engadir ceros a todo elemento que caia "fóra" da matriz ata que esta teña dimensión maior ou igual á do filtro. Tamén se utiliza recheo cando non se quere que a matriz de saída reduza o tamaño con respecto á matriz de entrada (Lopez Pinaya *et al.*, 2019). Existen tres tipos principais de recheo:

1. **Recheo válido (*valid padding*)**: Tamén se coñece como *no padding*, ou non recheo. A última convolución descártase se as dimensións non son válidas.

2. **Mesmo recheo (*same padding*)**: Este recheo asegura que a capa de saída terá a mesma dimensión que a de entrada.

3. **Recheo completo (*full padding*)**: Aumenta o tamaño da saída engadindo ceros ao borde da entrada de modo que a cada píxel se lle aplique o mesmo número de veces o filtro.

Podemos ver unha comparativa de recheo completo e mesmo recheo na Figura 2.6.

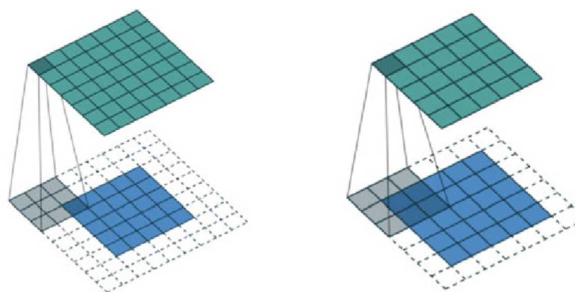
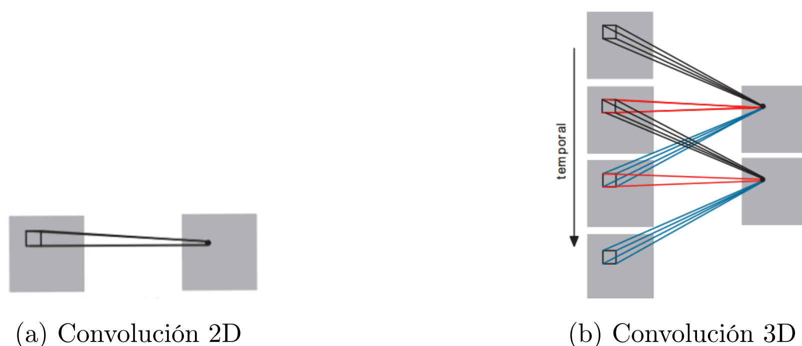


Figura 2.6: Recheo completo (á esquerda) e mesmo recheo (á dereita) (Kumar *et al.*, 2020).

Como a matriz de saída non está directamente conectada con cada elemento de entrada, dise que as capas convolucionais e de *pooling* están "parcialmente conectadas", aínda que esta característica tamén se pode chamar conectividade local.

O caso xeral é o que se mostra no exemplo, pero tamén se poden facer convolucións dunha dimensión, moi utilizadas en análise de sons e sinais, ou de máis dimensións, como por exemplo a análise de vídeos con convolucións tridimensionais (Ji *et al.*, 2013; Baccouche *et al.*, 2011). Na Figura 2.7 podemos observar unha comparación entre unha convolución 2D e unha 3D. Na Figura 2.7b, o tamaño do filtro de convolución na dimensión temporal é 3 e os conxuntos de conexións están indicados en diferentes cores de forma que os pesos compartidos teñen a mesma cor. Nas convolucións 3D, o mesmo filtro 3D é aplicado a cubos 3D solapados no vídeo de entrada para extraer características de movemento.



(a) Convolución 2D

(b) Convolución 3D

Figura 2.7: Comparación entre convolución 2D e 3D (Ji *et al.*, 2013).

Tamén existen outras formas de analizar vídeos con redes neuronais mediante o adestramento de diferentes redes para o tempo e o espazo, combinando as características obtidas en cada unha delas, como fan Karpathy *et al.* (2014) e Simonyan & Zisserman (2014).

Tras cada operación de convolución, a rede convolucional aplica unha transformación ReLU ao mapa de características para engadir non linealidade ao modelo.

Ademais do *zero-padding*, hai dous hiperparámetros que deben ser decididos antes do adestramento da rede: o número de filtros aplicado, que afectará á profundidade da saída (tres filtros producen tres mapas de características, unha saída de profundidade 3), e o *stride* ou paso, a distancia (número de píxeis) que o kernel se move sobre a matriz de entrada. Non é habitual ter pasos de tamaño dous ou máis, pero canto máis grande sexa o paso, máis pequena será a saída. Un tamaño de paso maior permite centrarse en estruturas de máis alto nivel e non tanto en detalles máis enlazados coa posición.

Un gran problema coas capas convolucionais é que o mapa de características é dependente da posición. Durante o adestramento, as CNN aprenden a asociar a presencia dunha certa característica cunha posición específica na imaxe de entrada, o que pode empobrecer o resultado (Kirsch, 2021). Porén, como diximos anteriormente, as redes convolucionais son equivariantes, e isto débese ás capas de *pooling*.

Capa de *pooling*

Tamén chamada capa de *downsampling* (submostraxe, redución de mostraxe ou redución de resolución), executa unha redución de dimensionalidade, reducindo o número de parámetros da entrada. De maneira similar á capa convolucional, a operación de *pooling* pasa un filtro a través de toda a entrada, coa diferencia de que este filtro non ten pesos (os valores do filtro na convolución), senón que aplica unha función de agregación aos valores do campo receptivo para completar a matriz de saída. Existen dous tipos principais:

1. ***Max pooling***: Segundo o filtro se move a través da entrada, selecciona o valor máximo e o envía á matriz de saída. É o tipo de *pooling* máis utilizado. Formalmente, sendo a matriz de entrada A de dimensión $n \times m$ e o filtro de dimensión $k \times l$:

$$v_{g,h} = \text{máx}\{a_{i,j} \mid k(g-1)+1 \leq i \leq kg \ \text{ e } \ l(h-1)+1 \leq j \leq lh\},$$

   con $g \in \{1, \dots, \lfloor \frac{n}{k} \rfloor\}$ e $h \in \{1, \dots, \lfloor \frac{m}{l} \rfloor\}$.

2. ***Average pooling***: Calcula o valor medio redondeado de cada campo receptivo para enviarllo á matriz de saída. Do mesmo modo que antes, formalmente:

$$v_{g,h} = \frac{\sum_{i=k(g-1)+1}^{kg} \sum_{j=l(h-1)+1}^{lh} a_{i,j}}{k+l},$$

   con $g \in \{1, \dots, \lfloor \frac{n}{k} \rfloor\}$ e $h \in \{1, \dots, \lfloor \frac{m}{l} \rfloor\}$.

A pesar de que se perde moita información ao utilizar esta técnica, ten os beneficios de que reduce a complexidade, mellora a eficiencia e reduce o risco de realizar un sobreaxuste. Isto

permite que a rede teña certa tolerancia a pequenas perturbacións aos datos de entrada, como por exemplo que dúas imaxes sexan case idénticas salvo por un desplazamento lateral de algúns píxeis (equivarianza).

Exemplo 2.14. Sexa $A \in \mathbb{R}^{4\times 4}$, definida como segue, a matriz de entrada dunha capa de *pooling* nunha rede convolucional

$$A = \begin{pmatrix} 1 & 1 & 0 & 2 \\ 4 & 5 & 1 & 0 \\ 1 & 2 & 3 & 0 \\ 0 & 1 & 1 & 4 \end{pmatrix},$$

e sexa $F \in \mathbb{R}^{2\times 2}$ un filtro *average pooling*[4] desa mesma capa. Aplicando o filtro á matriz cun paso de tamaño 2, obtemos:

$$v_{1,1} = \lfloor (1+1+4+5)/4 \rceil = 3, \qquad v_{1,2} = \lfloor (0+2+1+0)/4 \rceil = 1,$$

$$v_{2,1} = \lfloor (1+2+0+1)/4 \rceil = 1, \qquad v_{2,2} = \lfloor (3+0+1+4)/4 \rceil = 2,$$

o que se traduce na matriz de saída $v = \begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix}$. □

Exemplo 2.15. Sexa $A \in \mathbb{R}^{4\times 4}$, definida como segue, a matriz de entrada dunha capa de *pooling* nunha rede convolucional

$$A = \begin{pmatrix} 1 & 1 & 0 & 2 \\ 4 & 5 & 1 & 0 \\ 1 & 2 & 3 & 0 \\ 0 & 1 & 1 & 4 \end{pmatrix},$$

e sexa $F \in \mathbb{R}^{2\times 2}$ un filtro *max pooling* desa mesma capa. Aplicando o filtro á matriz cun paso de tamaño 2, obtemos:

$$v_{1,1} = \text{máx}\{1,1,4,5\} = 5, \qquad v_{1,2} = \text{máx}\{0,2,1,0\} = 2,$$

$$v_{2,1} = \text{máx}\{1,2,0,1\} = 2, \qquad v_{2,2} = \text{máx}\{3,0,1,4\} = 4,$$

o que se traduce na matriz de saída $v = \begin{pmatrix} 5 & 2 \\ 2 & 4 \end{pmatrix}$. □

[4]A operación de redondeo indicarase co símbolo $\lfloor x \rceil$, onde $\lfloor x \rceil = \begin{cases} \lceil x \rceil & \text{se} \quad x - \lfloor x \rfloor \geq 0{,}5, \\ \lfloor x \rfloor & \text{se} \quad x - \lfloor x \rfloor < 0{,}5. \end{cases}$

Capa completamente conectada

Anteriormente indicamos que nas capas parcialmente conectadas os valores dos píxeis de entrada non están directamente conectados coa capa de saída. Porén, na capa completamente conectada (tamén chamada capa densa), cada nodo da capa de saída está directamente conectado cun nodo da capa previa (IBM, 2020c).

Esta capa realiza a tarefa de clasificación baseada nas características extraídas nas capas anteriores. As FCL utilizan habitualmente a función *softmax* para clasificar as entradas, producindo unha probabilidade de entre 0 e 1.

2.3. Funcións de custo e perda

Sabemos que as redes neuronais pretenden imitar ao cerebro humano e, por tanto, pretenden aprender para mellorar os seus resultados, pero, como se avalía esta mellora? A resposta dáse cunha **función de perda**, que cuantifica a diferenza entre a saída esperada e a saída producida polo modelo. Denotaremos a partir de agora por $\hat{y}$ **o valor predito** pola rede neuronal e por y **o valor real** asociados a unha **entrada** x. A maioría dos algoritmos de aprendizaxe consisten en **minimizar unha función de custo** (Bottou, 1991) da forma:

$$C(y, \hat{y}) = \mathbb{E}(J(y, \hat{y})),$$

onde $\mathbb{E}$ denota a esperanza matemática a cal depende da función de probabilidade que caracteriza o problema a aprender, P, que é descoñecida, e J a función de perda que define o sistema de aprendizaxe. Xa que non para todos os problemas é posible ou sinxelo calcular a súa distribución de probabilidade, o cálculo do custo realízase habitualmente aplicando certas operacións sobre as funcións de perda para cada par de saídas esperada e producida. Dependendo do obxectivo da rede, empregaranse funcións de perda e custo distintas. Dúas funcións de perda habituais son a función 0-1 e a función de perda cadrática, definidas a continuación.

Definición 2.16. Dados $y, \hat{y} \in \mathbb{R}$, a función de perda 0-1 vén dada por:

$$J(y, \tilde{y}) = \begin{cases} 1 & \text{se} \quad y \neq \hat{y}, \\ 0 & \text{noutro caso.} \end{cases}$$

Definición 2.17. Dados $y, \hat{y} \in \mathbb{R}$ e $k \in \mathbb{R}$ unha constante, a función de perda cadrática vén dada por:

$$J(y, \hat{y}) = k(y - \hat{y})^2.$$

Nótese que habitualmente se escolle $k = \frac{1}{2}$ para simplificar a derivada desta función.

Estas funcións tamén serven para avaliar a perda cando as predicións son vectores, modificando a perda cadrática por medio do uso da norma euclidiana. Por outro lado, algunhas das funcións de custo máis habituais son o MSE (*Mean Squared Error*) ou erro cadrático medio (Aggarwal, 2018), o MAE (*Mean Absolute Error*) ou erro medio absoluto (Zhang & Sabuncu, 2018), a función de verosimilitude (*likelihood function*) e a función de entropía cruzada (*cross-entropy*), que definimos a continuación.

Definición 2.18. Sexan $\hat{y}_i, y_i \in \mathbb{R}$, $i \in \{1, \dots, n\}$, $n \in \mathbb{N}$, conxuntos de predicións dun modelo e os seus valores reais asociados. Entón a función do erro cadrático medio vén dada por:

$$C(y, \hat{y}) = \text{MSE}(y, \hat{y}) = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n}.$$

Definición 2.19. Sexan $\hat{y}_i, y_i \in \mathbb{R}$, $i \in \{1, \dots, n\}$, $n \in \mathbb{N}$, conxuntos de predicións dun modelo e os seus valores reais asociados. Entón a función do erro medio absoluto vén dada por:

$$C(y, \hat{y}) = \text{MAE}(y, \hat{y}) = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n}.$$

Para predicións de vectores, a MSE e MAE modifícanse engadindo a consideración da norma euclídea.

Definición 2.20. Sexan $\hat{y}_i \in (0, 1), y_i \in \{0, 1\}$, $i \in \{1, \dots, n\}$, $n \in \mathbb{N}$, conxuntos de predicións dun modelo e os seus valores reais asociados. Entón a función de verosimilitude vén dada por:

$$C(y, \hat{y}) = \prod_{i=1}^{n} \left(\hat{y}_i^{y_i} (1 - \hat{y}_i)^{1 - y_i} \right).$$

Neste caso trátase de maximizar a función de verosimilitude (á que habitualmente se lle aplica o logaritmo xa que é unha función monótona crecente), ou o que é equivalente, minimizar a función de entropía cruzada.

Definición 2.21. Sexan $\hat{y}_i \in (0, 1), y_i \in \{0, 1\}$, $i \in \{1, \dots, n\}$, $n \in \mathbb{N}$, conxuntos de predicións dun modelo e os seus valores reais asociados. Entón a función de entropía cruzada vén dada por:

$$C(y, \hat{y}) = -\sum_{i=1}^{n} \left(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right).$$

As dúas últimas funcións son útiles para clasificación binaria e, cando se trata de vectores, clasificación binaria múltiple ou, equivalentemente, clasificación multiclase con clases non excluíntes. Para avaliar o custo da clasificación binaria múltiple engádese un produtorio e sumatorio, respectivamente, sobre os elementos do vector nas funcións anteriores, é dicir:

$$C(y, \hat{y}) = \prod_{j=1}^{m} \prod_{i=1}^{n} \left((\hat{y}_i^j)^{y_i^j} (1 - \hat{y}_i^j)^{1 - y_i^j} \right),$$

no caso da función de verosimilitude, con $m \in \mathbb{N}$, e

$$C(y, \hat{y}) = -\sum_{j=1}^{m}\sum_{i=1}^{n}\left(y_i^j \log(\hat{y}_i^j) + (1 - y_i^j)\log(1 - \hat{y}_i^j)\right),$$

no caso da función de entropía cruzada.

Cada elemento é unha probabilidade entre 0 e 1 para o vector predito e exactamente 0 ou 1 para o vector real. Para poder avaliar clasificación multiclase con clases excluíntes, onde o vector real terá todo ceros salvo nunha posición, emprégase unha función chamada entropía cruzada categórica (Zhang & Sabuncu, 2018).

Definición 2.22. Sexan $\hat{y} = \{\hat{y}^1, \dots, \hat{y}^n\}$, $\hat{y}^i \in (0, 1)$ o vector de probabilidades predito pola rede e $y = \{0, \dots, y^i, \dots 0\}$, $y^i = 1$ o vector de clasificación real asociado, $i \in \{1, \dots, n\}$, $n \in \mathbb{N}$. Entón a función de entropía cruzada categórica vén dada por:

$$C(y, \hat{y}) = -\sum_{i=1}^{n} y^i \log(\hat{y}^i).$$

2.4. Algoritmos de aprendizaxe

O adestramento dunha rede neuronal consiste en amosarlle pares de entrada e saída de exemplo do problema para que poida optimizar a función de custo. Para optimizar esta función empréganse os chamados algoritmos de aprendizaxe (como por exemplo o descenso de gradiente estocástico) que axustan os pesos das diferentes capas da rede ata acadar, idealmente, o seu óptimo global. A referencia principal utilizada nesta sección é Ruder (2016).

2.4.1. Descenso de gradiente

Un dos algoritmos de aprendizaxe máis coñecidos e utilizados é o **descenso de gradiente** (*gradient descent*), xunto coas súas variantes. Este algoritmo consiste en actualizar os valores dos pesos da rede en proporción ao gradiente da función de custo, empregando todo o conxunto de adestramento, con respecto aos pesos para todo o conxunto de datos, é dicir:

$$w_{t+1} = w_t - \eta \nabla_{w_t} C(y, \hat{y}).$$

Para decidir cantas actualizacións se realizan aos pesos, decídese un número de **épocas** (*epochs*) para o adestramento, é dicir, o número de iteracións que realiza o algoritmo de aprendizaxe sobre o conxunto de datos de adestramento[5]. A taxa de aprendizaxe ($\eta \in (0, \infty)$ na ecuación

[5] En ocasións emprégase unha técnica coñecida como **parada anticipada** (*early stopping*) que finaliza o adestramento antes de realizar todas as iteracións para evitar o sobreaxuste.

anterior) determina como de grande é a actualización realizada. A descrición algorítmica deste modelo, en pseudo-código, sería como segue:

Pseudocódigo descenso de gradiente

```
for i in num_epocas
    gradiente = avaliar_gradiente(funcion_custo, datos, pesos)
    pesos = pesos - taxa_aprendizaxe * gradiente
```

O descenso de gradiente garante a converxencia a un óptimo global para superficies de erro, ou custo, convexas, pero só garantiza a converxencia a un mínimo local en superficies non convexas.

No caso de modelos compostos por unha soa capa de neuronas, o cálculo do gradiente para a optimización da función de custo é sinxelo, xa que é función directa dos pesos da capa. Porén, este cálculo resulta moito máis complexo cando hai múltiples capas, cadansúa cos seus pesos. Para poder realizar este cálculo emprégase o algoritmo de **retropropagación** (Werbos, 1974; Rumelhart *et al.*, 1986), que se explicará nesta sección.

A actualización no descenso de gradiente tradicional realízase tras computar o custo para todos os valores do conxunto de adestramento (o que se denomina **lote**, ou *batch*[6]), que é moi lento e, nos casos nos que o tamaño do conxunto é maior ao espazo libre en memoria, intratable. Por este motivo se desenvolveron variantes deste algoritmo, que presentamos a continuación.

Descenso de gradiente estocástico

O **descenso de gradiente estocástico** (*stochastic gradient descent*, SGD) é unha variación do descenso de gradiente que actualiza o valor dos pesos tras avaliar cada par entrada-saída:

$$w_{t+1} = w_t - \eta \nabla_{w_t} C(y_i, \hat{y}_i).$$

Habitualmente os cambios realizados nos valores dos pesos teñen unha varianza moi alta, o que resulta en grandes flutuacións na función de custo. Por un lado, isto permite que o algoritmo non quede "entorado" nun mínimo local pero, por outro lado, dificulta a súa converxencia ao óptimo global. Non obstante, a súa converxencia é equiparable á do descenso de gradiente cando se reduce a taxa de aprendizaxe lentamente. Como se ve na súa descrición algorítmica, amosada a continuación, mestúrase o conxunto de datos de adestramento en cada época para que as actualizacións sexan máis efectivas e non se repitan sempre na mesma orde.

[6]Na literatura denomínase ás veces este método como descenso de gradiente por lotes (*batch gradient descent*).

Pseudocódigo descenso de gradiente estocástico

```
for i in num_epocas
    mesturar(datos)
    for elemento in datos
        gradiente = avaliar_gradiente(funcion_custo, elemento, pesos)
        pesos = pesos - taxa_aprendizaxe * gradiente
```

A pesar de ser moito máis rápido que o descenso de gradiente tradicional, realizar actualizacións para cada elemento do conxunto de adestramento ten un gran custo computacional, polo que se definiu unha solución intermedia.

Descenso de gradiente por mini lotes

Esta solución intermedia é o **descenso de gradiente por mini lotes** (*mini-batch gradient descent*), que utiliza lotes máis pequenos denominados **mini lotes**, de modo que os pesos se actualizan unha vez procesado cada mini lote. O adestramento finaliza tras avaliar todos os mini lotes. Este método ten un menor custo computacional que o descenso de gradiente estocástico e permite máis actualizacións dos valores dos pesos que o tradicional. Ademais, reduce a varianza das actualizacións dos parámetros que observamos no descenso estocástico, o que estabiliza a súa converxencia. Na seguinte ecuación k denota o tamaño do mini lote.

$$w_{t+1} = w_t - \eta \nabla_{w_t} C(y_{\{i,\dots,i+k\}}, \hat{y}_{\{i,\dots,i+k\}}),$$

con $i \in \{1, k+1, 2k+1, \dots, n-k\}$.

A súa descrición algorítmica en pseudo-código amósase a continuación. Do mesmo modo que no descenso estocástico, mestúranse os datos antes de dividilos en lotes.

Pseudocódigo descenso de gradiente por mini lotes

```
for i in num_epocas
    mesturar(datos)
    for lote in dividir_lotes(datos, tamaño_lote)
        gradiente = avaliar_gradiente(funcion_custo, lote, pesos)
        pesos = pesos - taxa_aprendizaxe * gradiente
```

Habitualmente na literatura denomínase ao descenso de gradiente por mini lotes como descenso estocástico ou SGD, a pesar de ser diferentes variantes.

2.4.2. Retropropagación

As predicións realizadas por redes multicapa son, en esencia, composición das funcións de cada unha das súas capas. O algoritmo de retropropagación aproveita a regra da cadea (de cálculo diferencial) para poder calcular o gradiente da función de custo composta en termos de sumas dos produtos de gradientes locais nos diferentes camiños que se poden percorrer dende un nodo ata a saída (Aggarwal, 2018). Para poder computar de maneira eficiente o número de camiños que compoñen a suma emprégase **programación dinámica**. O algoritmo de retropropagación é unha aplicación directa deste tipo de programación e está composto por dúas fases principais:

1. **Fase de avance** (*forward phase*): Envíanse as entradas á rede, utilizando os pesos actuais para os diferentes cálculos. Compárase a saída obtida polo modelo coa saída esperada e calcúlase a derivada da función de custo con respecto da saída.

2. **Fase de retroceso** (*backward phase*): Nesta fase trátase de atopar o gradiente da función de custo con respecto a cada un dos pesos mediante a regra da cadea e utilizalo para actualizar os pesos. Dado que o gradiente se computa cara atrás dende a última capa, a fase denomínase "de retroceso".

Seguindo a explicación realizada en Aggarwal (2018), supoñamos que para un certo modelo con neuronas u_i, $i \in \{1, \dots, n\}$, u_0 a entrada da rede, $w_{(u_i, u_{i+1})}$ os pesos asociados aos seus enlaces, C a súa función de custo e o a súa saída, só hai un camiño dende u_1 ata o. Entón temos que o gradiente correspondente a calquera dos pesos vén dado por:

$$\frac{\partial C}{\partial w_{(u_{r-1}, u_r)}} = \frac{\partial C}{\partial o} \cdot \left[\frac{\partial o}{\partial u_n} \prod_{i=r}^{n-1} \frac{\partial u_{i+1}}{\partial u_i} \right] \frac{\partial u_r}{\partial w_{(u_{r-1}, u_r)}} \quad \forall r \in \{1, \dots, n\}.$$

Existe unha variante xeneralizada da regra da cadea, a **regra da cadea multivariante**, que nos permite calcular o gradiente cando hai múltiples camiños dende cada unha das unidades ata a saída. Entón, mantendo a notación anterior para un set $\mathcal{P}$ de camiños dende u_r ata o, o gradiente para calquera dos pesos vén dado como segue:

$$\frac{\partial C}{\partial w_{(u_{r-1}, u_r)}} = \frac{\partial C}{\partial o} \cdot \left[\sum_{[u_r, \dots, u_n, o] \in \mathcal{P}} \frac{\partial o}{\partial u_n} \prod_{i=r}^{n-1} \frac{\partial u_{i+1}}{\partial u_i} \right] \frac{\partial u_r}{\partial w_{(u_{r-1}, u_r)}} \quad \forall r \in \{1, \dots, n\}. \tag{2.1}$$

O algoritmo de retropropagación computa todo o lado dereito da ecuación agás $\frac{\partial u_r}{\partial w_{(u_{r-1}, u_r)}}$. Esta computación (que se denotará por $\Delta(u_r, o) = \frac{\partial C}{\partial u_r}$) é máis sinxela de tratar con este algoritmo, xa que se avalía recursivamente o seu valor dende o último nodo e, por tanto, para un u_r dado, teriamos o valor correspondente a todos os nodos seguintes, dende u_{r+1} ata o, inicializando

$\Delta(o, o) = \frac{\partial C}{\partial o}$. Temos entón:

$$\Delta(u_r, o) = \frac{\partial C}{\partial u_r} = \sum_{u|u_r \Rightarrow u} \frac{\partial C}{\partial u} \frac{\partial u}{\partial u_r} = \sum_{u|u_r \Rightarrow u} \frac{\partial u}{\partial u_r} \Delta(u, o).$$

Coñecendo o valor de $\Delta(u, o)$, só queda calcular $\frac{\partial u}{\partial u_r}$. Sexa z_u o valor obtido tras aplicar a función de entrada da unidade u e f a súa función de activación, entón $u = f(z_u)$ e temos:

$$\frac{\partial u}{\partial u_r} = \frac{\partial u}{\partial z_u} \frac{\partial z_u}{\partial u_r} = \frac{\partial f(z_u)}{\partial z_u} w_{(u_r,u)} = f'(z_u) w_{(u_r,u)}.$$

Retornando á ecuación anterior e substituíndo este valor, obtemos:

$$\Delta(u_r, o) = \sum_{u|u_r \Rightarrow u} f'(z_u) w_{(u_r,u)} \Delta(u, o).$$

Finalmente, o único valor que queda coñecer para poder resolver a Ecuación (2.1) é $\frac{\partial u_r}{\partial w_{(u_{r-1},u_r)}}$, que utilizando a mesma notación resulta:

$$\frac{\partial u_r}{\partial w_{(u_{r-1},u_r)}} = u_{r-1} f'(z_{u_r}).$$

O gradiente clave que se propaga cara atrás é a derivada con respecto da función de activación das capas e o gradiente con respecto aos pesos é fácil de computar para todos os enlaces dunha unidade correspondente.

Para poder empregar o algoritmo de retropropagación, non obstante, requírese que a función que compón a rede, é dicir, a composición das diferentes funcións de entrada, saída, activación e custo que compoñen as capas, sexa diferenciable por partes xa que, como vemos nas ecuacións, é necesario coñecer as derivadas parciais de cada unha destas funcións.

Retropropagación a través do tempo

Como xa explicamos anteriormente, as redes recorrentes teñen certas variacións con respecto ás redes de avance cara diante. Principalmente, engaden un estado oculto que retroalimenta á rede e os pesos de cada "capa temporal"[7] son compartidos en toda a rede. Para poder xestionar os pesos compartidos creouse unha variación do algoritmo de retropropagación, chamada **retropropagación a través do tempo** (Werbos, 1990).

Inicialmente debe supoñerse que os pesos dunha capa temporal son independentes para cada paso temporal t, denotados polas variables temporais W^t_{xh}, W^t_{hh} e W^t_{hy}. Baixo este suposto podemos aplicar o algoritmo de retropropagación habitual, calculando o gradiente para cada

[7]Referímonos a cada unha das diferentes capas que aparecen nunha representación desenrolada dunha rede recorrente como "capa temporal".

unha das variables temporais e, posteriormente, engadindo as súas contribucións individuais ao gradiente do peso compartido (Aggarwal, 2018). É dicir, compútanse as seguintes ecuacións:

$$\frac{\partial C}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial C}{\partial W_{xh}^{t}}, \qquad \frac{\partial C}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial C}{\partial W_{hh}^{t}}, \qquad \frac{\partial C}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial C}{\partial W_{hy}^{t}}.$$

Nestas ecuacións W_{xh} denota á matriz de pesos dos enlaces dende a entrada ata unha capa oculta, W_{hh} a dos enlaces entre capas ocultas, W_{hy} a dos enlaces dende a última capa oculta á saída e $T \in \mathbb{N}$ o número de pasos temporais avaliados.

2.4.3. Métodos de optimización

Os algoritmos de aprendizaxe descritos nesta sección teñen certas dificultades. A escolla dunha taxa de aprendizaxe axeitada pode ser complexa, xa que se é moi pequena a converxencia pode ser moi lenta pero, se é demasiado grande, pode interferir coa converxencia aumentando a fluctuación da función de custo. Existen "programas" de taxa de aprendizaxe (*learning rate schedules*) que tratan de axustala no adestramento segundo unha programación predefinida (Robbins & Monro, 1951) ou cando a función de custo baixa dun valor predeterminado, pero estes programas teñen que decidirse de antemán e por tanto non son capaces de adaptarse ás características concretas dun conxunto de datos en particular (Darken *et al.*, 1992). Por outro lado, a taxa aplícase a todos os parámetros por igual pero, se as características dos datos teñen frecuencias moi diferentes, podemos non querer que se actualicen todos na mesma proporción. Por último, un dos problemas máis comúns é a escolla de mínimos locais que non son óptimos globais en funcións con superficies non convexas. Na publicación de Dauphin *et al.* (2014) indícase que a problemática está nos puntos de sela (*saddle points*) e non nos mínimos locais, que están habitualmente rodeados dunha meseta do mesmo valor para a función de custo, o que fai complicado que o SGD escape, ao ser o gradiente case cero en todas as dimensións. Para superar estas dificultades existen múltiples métodos de optimización do descenso de gradiente. Entre eles podemos atopar os algoritmos do momento, o NAG, o Adagrad, o RMSprop e o Adam.

Momento

O descenso estocástico ten dificultades para tratar "barrancos" (*ravines*), é dicir, áreas que curvan moito máis abruptamente nunha dimensión que en outras (Sutton, 1986), comúns en óptimos locais. O **método do momento** (Qian, 1999) é un método de optimización que acelera o proceso do SGD reducindo o número de oscilacións que realiza ata chegar ao óptimo neste tipo de áreas. Vemos unha comparación de SGD sen momento e con momento na Figura 2.8.

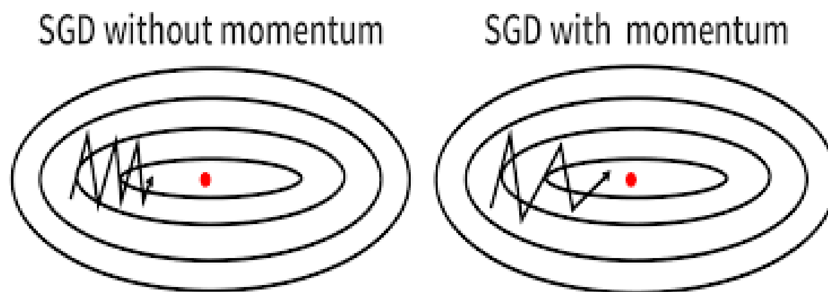


Figura 2.8: Comparativa de SGD con momento e sen momento (Du, 2019).

Para acelerar o descenso de gradiente na dirección do óptimo, o método do momento engade un termo adicional na actualización dos pesos. É dicir, en lugar de $w_{t+1} = w_t - \eta \nabla_{w_t} C(y, \hat{y})$, este método actualiza o peso mediante a ecuación

$$w_{t+1} = w_t - v_t = w_t - (\gamma v_{t-1} + \eta \nabla_{w_t} C(y, \hat{y})),$$

onde o termo $v_t = \gamma v_{t-1} + \eta \nabla_{w_t} C(y, \hat{y})$ se denomina **momento temporal**, e $v_0 = 0$. Habitualmente escóllese $\gamma = 0{,}9$.

Para ilustrar o concepto, equiparemos o descenso de gradiente a unha bóla baixando por un val. O método do momento engade ao descenso tradicional os conceptos de velocidade (o momento temporal) e fricción. O gradiente actúa como unha forza que modifica a velocidade á que descende á bóla e a fricción tende a reducir a velocidade gradualmente. Se os gradientes en varias iteracións para un certo peso están (aproximadamente) na mesma dirección, a velocidade aumentará, mentres que se cambia de dirección, o termo de fricción (γv_{t-1}) causará que a velocidade diminúa (Nielsen, 2015).

NAG

O método do **gradiente acelerado de Nesterov** (*Nesterov accelerated gradient*, NAG) é unha mellora do método do momento (Nesterov, 1983) que calcula o gradiente da función de custo pero tendo en conta a seguinte posición aproximada dos parámetros grazas ao momento temporal. Para reflectir este cambio, utilizarase un subíndice na función de custo $C_{w_t - \gamma v_{t-1}}$, como vemos na ecuación

$$w_{t+1} = w_t - v_t = w_t - (\gamma v_{t-1} + \eta \nabla_{w_t} C_{w_t - \gamma v_{t-1}}(y, \hat{y})),$$

onde $v_t = \gamma v_{t-1} + \eta \nabla_{w_t} C_{w_t - \gamma v_{t-1}}(y, \hat{y})$.

Desta forma evitamos acelerar demasiado rápido e aumentamos a eficiencia do descenso de gradiente.

Adagrad

Adagrad (Duchi *et al.*, 2011) é un algoritmo que adapta a taxa de aprendizaxe aos parámetros, realizando actualizacións máis grandes para parámetros asociados a características infrecuentes (as características j tal que na maioría dos exemplos de entrada $x_j = 0$, $j \in \{1, \dots, k\}$, $k \in \mathbb{N}$) e máis pequenas para parámetros asociados a características frecuentes, polo que é moi axeitado para tratar datos dispersos.

Os métodos anteriores realizaban a actualización para todos os parámetros á vez, ao utilizar a mesma taxa de aprendizaxe para todos eles. Adagrad, en cambio, utiliza unha taxa diferente para cada parámetro w_i, $i \in \{1, \dots, n\}$, $n \in \mathbb{N}$, en cada paso temporal t, $t \in \{1, \dots, T\}$, $T \in \mathbb{N}$. En primeiro lugar, axustamos a ecuación de descenso de gradiente para incorporar o parámetro w_i, utilizando a mesma notación de subíndice para a función de custo que no método NAG:

$$w_{t+1,i} = w_{t,i} - \eta g_{t,i} = w_{t,i} - \eta \nabla_{w_{t,i}} C_{w_{t,i}}(y, \hat{y}),$$

onde $g_{t,i} = \nabla_{w_{t,i}} C_{w_{t,i}}(y, \hat{y})$ é o gradiente da función de custo con respecto a $w_{t,i}$.

Nas actualizacións, Adagrad modifica a taxa de aprendizaxe xeral η en cada paso temporal t para cada parámetro w_i baseándose nos anteriores gradientes calculados para ese parámetro:

$$w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}.$$

Nesta ecuación, $G_t \in \mathbb{R}^{n \times n}$ é unha matriz diagonal onde cada elemento ii é a suma dos cadrados dos gradientes con respecto a w_i ata o paso temporal t, sen incluír, e ϵ é un termo de suavizado que evita as divisións entre 0 (soe tomarse $\epsilon = 10^{-8}$). Podemos vectorizar esta ecuación definindo o vector $K_t = \left(k_i = \frac{1}{\sqrt{G_{t,ii}+\epsilon}} \right) \in \mathbb{R}^n$ de modo que:

$$w_{t+1} = w_t - \eta K_t \circ g_t, \tag{2.2}$$

onde $\circ$ representa o produto Hadamard, w_t o vector dos pesos $w_{t,i}$ e g_t o vector dos gradientes $g_{t,i}$.

A vantaxe principal deste método é que non é necesario o axuste manual da taxa de aprendizaxe. Porén, ten a desvantaxe de que se vai reducindo ata volverse infinitamente pequena debido á acumulación dos gradientes no denominador do vector K_t, o que causa que a partir dun certo punto o algoritmo deixe de aprender.

Finalmente, describimos os dous métodos de optimización máis utilizados nas redes neuronais: RMSprop e Adam.

RMSprop

RMSprop é un algoritmo non publicado proposto por Geoffrey Hinton que trata de solucionar a desvantaxe de Adagrad para a taxa de aprendizaxe (Ruder, 2016). Para iso en cada etapa t, tómase a suma ata a etapa $t-1$ dos gradientes ao cadrado (como no algoritmo anterior) multiplicada por 0,9 e súmaselle o cadrado do gradiente na etapa t multiplicado por 0,1, de modo que a media móbil, $\mathbb{E}\left[g^2\right]_{t,i}$, en t depende da media anterior e o gradiente actual, e utilízase para substituír a matriz diagonal que determina os elementos do vector K_t na Ecuación (2.2), é dicir:

$$\mathbb{E}\left[g^2\right]_{t,i} = \gamma\mathbb{E}\left[g^2\right]_{t-1,i} + (1-\gamma)g_{t,i}^2,$$

$$w_{t+1} = w_t - \eta K_t \circ g_t,$$

con $K_t = \left(k_i = \frac{1}{\sqrt{\mathbb{E}[g^2]_{t,i}+\epsilon}}\right) \in \mathbb{R}^n$ e $\gamma = 0{,}9$.

O denominador de cada compoñente do vector K_t na ecuación anterior é a media cadrática (*root mean squared*, RMS) do gradiente, salvo o ϵ, de onde toma o seu nome o algoritmo. Entón podemos reescribir a ecuación como

$$w_{t+1} = w_t - \frac{\eta}{\mathrm{RMS}(g)_t} \circ g_t,$$

onde se realiza un pequeno abuso de notación ao incorporar o vector $\mathrm{RMS}(g)_t$ no denominador, que representa ao vector $K_t = \left(k_i = \frac{1}{\mathrm{RMS}(g)_{t,i}}\right) \in \mathbb{R}^n$.

Hinton suxire que o valor da taxa de aprendizaxe η sexa 0,001.

Adam

Adam (*Adaptive Moment Estimation*) é outro método que adapta a taxa de aprendizaxe para cada parámetro (Kingma & Ba, 2015) que, ademais de almacenar a media decadente dos cadrados dos gradientes anteriores, almacena a media para os gradientes pasados. Sendo $v_t = \mathbb{E}\left[g^2\right]_t$ o vector formado polas medias decadentes dos cadrados dos gradientes anteriores para cada parámetro e $m_t = \mathbb{E}\left[g\right]_t$ o vector formado polas medias decadentes dos gradientes anteriores para cada parámetro, considera as ecuacións:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t,$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2.$$

O nome deste algoritmo vén das dúas ecuacións anteriores, xa que m_t e v_t son, respectivamente, estimacións do primeiro (a media) e segundo (a varianza non centrada) momentos dos gradientes. Os dous vectores son inicializados con ceros ($m_0 = (0, \dots, 0)$, $v_0 = (0, \dots, 0)$), polo

que teñen un certo sesgo cara o 0, especialmente nos primeiros pasos temporais ou cando as taxas de decaemento son pequenas (β_1 e β_2 próximos a 1). Para corrixir este sesgo computan as estimacións corrixidas dos momentos

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t},$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t},$$

onde β_k^t, $k \in \{1,2\}$, denota o parámetro β_k elevado á potencia t, e utilizan estas estimacións corrixidas para actualizar os parámetros:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \circ \hat{m}_t.$$

De novo realizamos un pequeno abuso de notación do mesmo modo que no apartado anterior ao incorporar o vector $\hat{v}_t$ no denominador.

Os autores suxiren os valores por defecto $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$ e $\epsilon = 10^{-8}$.

2.5. Hiperparámetros

Todas as redes neuronais teñen uns certos parámetros que teñen que ser decididos antes de comezar o adestramento. Entre eles poden estar, por exemplo, o tamaño do lote a utilizar, o número de neuronas que ten cada capa, a función de activación que empregar para unha determinada capa ou o método de optimización do algoritmo de aprendizaxe. Estes son chamados hiperparámetros (Goodfellow *et al.*, 2016), xa que a rede non adestra para melloralos como fai cos pesos (aínda que se pode deseñar un procedemento de aprendizaxe anidado onde un algoritmo aprende os mellores hiperparámetros para outro algoritmo de aprendizaxe). Ás veces selecciónase unha destas configuracións como hiperparámetro xa que é difícil de optimizar pero, habitualmente, débese a que non é apropiado que a rede o aprenda no conxunto de adestramento. Isto sucede cos hiperparámetros que controlan a capacidade do modelo. Se se aprenderan co conxunto de adestramento sempre se escollerían os hiperparámetros que desen a máxima capacidade ao modelo, o que implicaría un sobreaxuste. Por isto tamén é importante ter un conxunto de validación, como se indica no primeiro capítulo, para poder comprobar a xeneralización do modelo. En resumo, un hiperparámetro é un parámetro que ten que ser decidido antes de comezar o proceso de adestramento, axustable e que pode afectar directamente ao adestramento do modelo.

2.5.1. Optimización de hiperparámetros

Para decidir a mellor combinación de hiperparámetros para o modelo existen diferentes métodos. Por un lado está a **busca manual** (*manual tuning*), na que se selecciona unha combinación

de hiperparámetros manualmente e se realiza o adestramento con ela. Isto é moi laborioso xa que require tanto o coñecemento experto do funcionamento da rede como a especificación manual de que hiperparámetros se han de probar. Por outro lado, existen múltiples algoritmos de busca automática. Os máis populares son a busca aleatoria (*random search*), a busca exhaustiva (*grid search*) e a optimización bayesiana. Para os algoritmos de busca automática debe definirse un **espazo de busca**, é dicir, un conxunto de valores posibles para cada un dos hiperparámetros a optimizar.

A **busca exhaustiva** proba todas as combinacións de hiperparámetros posibles no espazo de busca e selecciona a que mellores resultados obtén para o modelo. Por exemplo, pódense seleccionar tres tamaños de lotes, dúas posibilidades para o número de neuronas e tres funcións de activación diferentes. A busca exhaustiva realizará entón o adestramento para $3 \cdot 2 \cdot 3 = 18$ combinacións diferentes. Como podemos comprobar, isto ten un alto custo computacional, especialmente para conxuntos de adestramento grandes.

A **busca aleatoria** consiste, como indica o seu nome, en escoller combinacións aleatorias de hiperparámetros dun espazo de busca en cada iteración coas que adestrar o modelo e seleccionar, tras finalizar todas as iteracións, a mellor combinación das probadas. Neste caso, o espazo de busca pode estar composto por conxuntos non finitos, xa que só se realizarán un número de iteracións predeterminadas.

Ambas as dúas técnicas anteriores non toman decisións informadas para a optimización dos hiperparámetros, senón que proban combinacións de hiperparámetros coa esperanza de que algunha obteña bos resultados. Pola contra, a **optimización bayesiana** ten en conta as decisións tomadas en pasos anteriores para a optimización dos hiperparámetros, o que mellora os resultados obtidos en comparación cos métodos anteriores (Hutter *et al.*, 2011; Bergstra *et al.*, 2011; Snoek *et al.*, 2012). Denomínase "bayesiana" xa que se basea no Teorema de Bayes:

Teorema 2.23 (Teorema de Bayes)**.** *Dados $A_1, \dots, A_n$, $n \in \mathbb{N}$ un conxunto de sucesos incompatibles dous a dous, cuxa unión é o suceso seguro e con probabilidade distinta de cero, se B é un suceso:*

$$P(A_i \mid B) = \frac{P(B \mid A_i) \cdot P(A_i)}{P(B)} = \frac{P(B \mid A_i) \cdot P(A_i)}{[P(B \mid A_1) \cdot P(A_1) + \dots + P(B \mid A_n) \cdot P(A_n)]},$$

con $i \in \{1, \dots, n\}$.

Axustando este resultado ao método de optimización bayesiana (Brochu *et al.*, 2010), temos que, en xeral, a probabilidade a posteriori dun modelo M dado un conxunto de observacións E realizadas é proporcional á probabilidade (*likelihood*) de E dado M multiplicado pola probabilidade a priori de M:

$$P(M|E) \propto P(E|M)P(M).$$

No noso caso, sexan x_i a i-ésima mostra e $f(x_i)$ a observación da función obxectivo (combinación das funcións de entrada, activación, saída, perda e custo) para x_i. Segundo acumulamos observacións, na iteración t temos $\mathcal{D}_t = \{(x_1, f(x_1)), ..., (x_t, f(x_t))\}$. A distribución a priori combínase coa función de probabilidade $P(\mathcal{D}_t|f)$. Entón, temos que a distribución a posteriori vén dada por:

$$P(f|\mathcal{D}_t) \propto P(\mathcal{D}_t|f)P(f).$$

A idea é que, na práctica, substitúese a función obxectivo, descoñecida ou custosa de avaliar, por outra máis sinxela (para poder avaliar os parámetros que interveñen no modelo) que á súa vez proporciona información sobre a probabilidade dos datos e vaise mellorando (achegando máis á real) a partir dos resultados do proceso de obtención de mostras.

Ata aquí unha breve presentación da optimización bayesiana. Se, adicionalmente, atopamos hiperparámetros na definición do obxectivo, lévase a cabo un proceso previo de optimización para atopar os valores óptimos de devanditos hiperparámemtros.

O método de optimización bayesiana equilibra a exploración (*exploration*) e a explotación (*exploitation*) da función obxectivo, é dicir, equilibra a escolla de valores para os hiperparámetros en zonas de alta incerteza e en zonas nas que é máis probable que haxa valores que melloren a mellor observación ata o momento. Desta forma, a optimización bayesiana trata de obter a maior cantidade de información posible sobre a función e onde pode estar o valor óptimo para mellorar a combinación de hiperparámetros escollida.

Capítulo 3

Aplicacións

Neste capítulo explicaremos un exemplo sinxelo realizado en Python que compara tres modelos diferentes e presentaremos varias aplicacións reais en múltiples campos da aprendizaxe profunda.

3.1. Exemplo en Python

Poñamos en primeiro lugar un exemplo sinxelo dunha rede neuronal codificada en **Python**, empregando a biblioteca **Keras**. O código correspondente a este exemplo pode atoparse no Anexo 3.2. Pretendemos crear un modelo que clasifique en dúas clases os puntos atopados nun círculo de radio 1 centrado no punto $(0, 0)$ e os puntos dun anel de radio interior 1,2 e radio exterior 2 centrado no mesmo punto. O conxunto de datos empregado no exemplo amósase na Figura 3.1 e está composto por 250 datos de cada unha das dúas clases posibles. Dado que as redes neuronais só traballan con valores numéricos, identificamos a categoría do círculo cun 0 e a do anel cun 1. A cada un dos datos de entrada asociámoslle o número da súa categoría correspondente como saída real.

Dividiuse este conxunto escollendo un 80 % dos datos para o adestramento e un 20 % para probas, mantendo a proporción da metade de elementos de cada clase. Para que o exemplo puidera ser replicable cos mesmos datos, empregamos unha "semente" para os valores e os elementos aleatorios dos modelos de 0 e xeramos os datos no círculo e anel empregando unha distribución uniforme para calcular as súas coordenadas polares, que transformamos antes de enviarllas ao modelo en coordenadas cartesianas.

Como vemos na imaxe, as dúas clases non son linealmente separables, polo que o uso dunha función lineal no modelo non debería poder aproximar correctamente a función que separa ambas

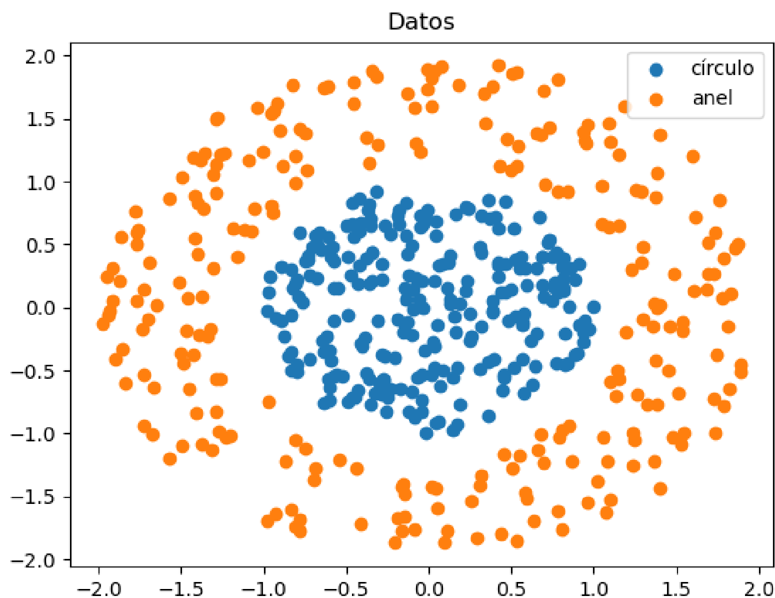


Figura 3.1: Datos utilizados para o exemplo.

categorías. Para comprobalo, creamos en primeiro lugar un modelo composto por unha soa capa densa con 5 neuronas e unha función de activación lineal ($f(z) = z$), empregando a función sigmoide como función de activación da capa de saída. Chamaremos a este modelo o **modelo lineal**. A continuación creamos un modelo idéntico pero esta vez empregando a función de activación ReLU en lugar da lineal, ao que denominaremos **modelo ReLU**. Por último, para comprobar o efecto que ten engadir máis capas, creamos outro modelo engadíndolle dúas capas máis idénticas ás do modelo anterior (densas, con 5 neuronas e función de activación ReLU), ao que chamaremos **modelo multicapa**. En todos os modelos se utilizou como función de custo a función de entropía cruzada e como algoritmo de aprendizaxe o descenso de gradiente estocástico.

Nas Figuras 3.2 e 3.3 podemos observar como o modelo lineal non é capaz de aprender ningunha función que aproxime a clasificación nas dúas categorías, como esperabamos, polo que a súa precisión e custo se manteñen case constantes. Podemos deducir da gráfica da súa precisión, preto de 0,5 para todas as épocas, que este modelo predí a mesma clase para case todos os valores xa que están distribuídos nun 50 % de datos de cada categoría e por tanto acerta en aproximadamente a metade dos datos. No conxunto de probas obtén unha precisión de case un 42 %, en comparación ao case 45 % que obtén no conxunto de adestramento. É habitual obter unha precisión lixeiramente menor para o conxunto de probas que para o conxunto de adestramento, como sucede neste caso.

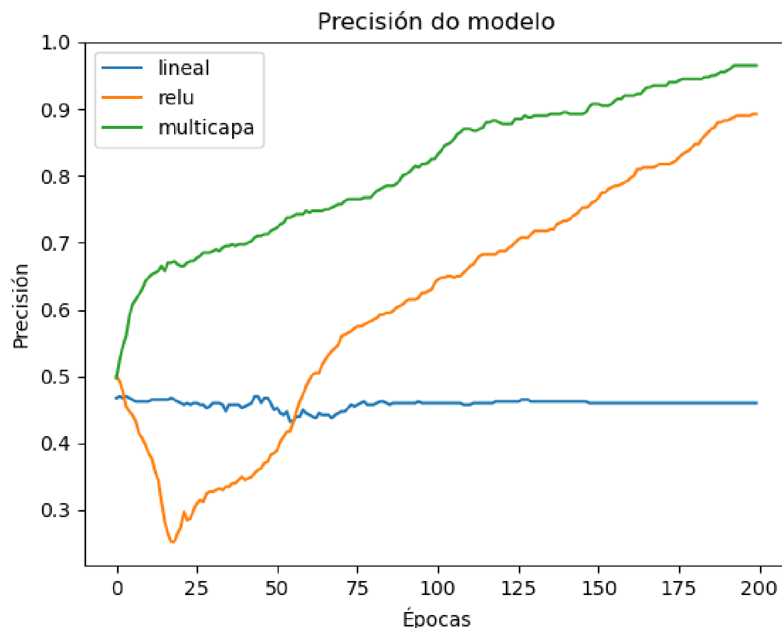


Figura 3.2: Comparativa da precisión dos tres modelos do exemplo.

Pola contra, o modelo ReLU ten unha tendencia a aumentar a súa precisión e diminuír o seu custo ao avanzar as épocas. Isto débese a que a función de activación ReLU é non lineal e por tanto aproxima mellor a función de clasificación que o modelo lineal. Se aumentaramos o número de épocas poderiamos ver unha converxencia a un valor concreto para ambas gráficas, xa que chegaría un punto no que a rede non sería capaz de aprender máis. A precisión deste modelo no conxunto de probas é de 88 % e de 89 % no conxunto de adestramento. Como era de esperar, este valor é case o dobre do que observamos no primeiro modelo, e mantense moi similar nos conxuntos de probas e adestramento. Isto indica que o modelo ten unha boa xeneralización e que non incorremos nin nun sobreaxuste nin nun subaxuste. De todos modos, tamén é habitual que a precisión en datos novos sexa aínda menor que en ambos conxuntos nos casos nos que se pode comprobar o seu valor, como é este exemplo.

Finalmente, podemos ver en ambas gráficas que o modelo multicapa mantén as mesmas tendencias que o modelo ReLU pero mellorando os seus valores. No conxunto de adestramento obtén unha precisión de 96 %, reducíndose a un 93 % no conxunto de probas. Como se indicou no capítulo anterior, a combinación de múltiples capas obtén mellores aproximacións que os modelos compostos por unha soa capa para funcións non lineais. Do mesmo modo que sucedía no modelo ReLU, se aumentásemos o número de épocas poderiamos observar unha converxencia da precisión e o custo.

En resumo, con este exemplo comprobamos como unha rede neuronal cunha soa función de

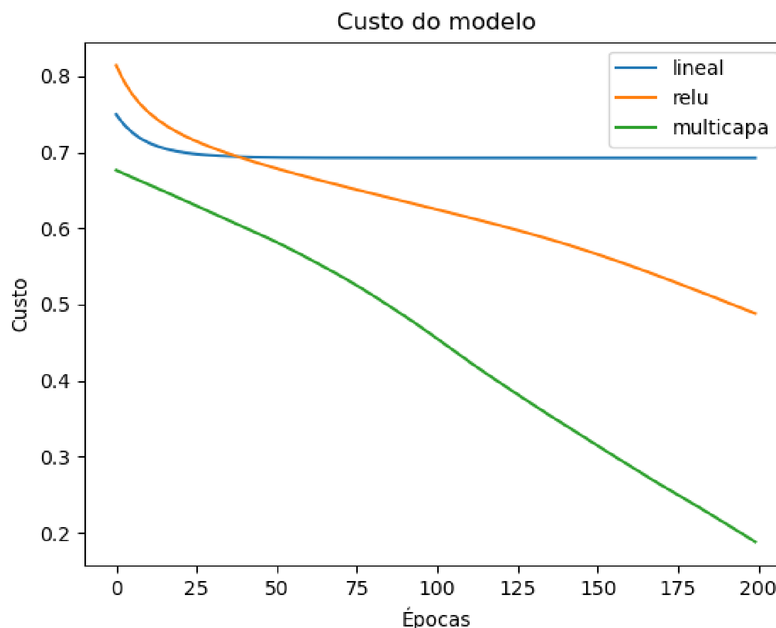


Figura 3.3: Comparativa do custo dos tres modelos do exemplo.

activación lineal non é capaz de aproximar unha función non lineal e que aumentando o número de capas con funcións de activación non lineais melloran os resultados obtidos polo modelo.

O exemplo que acabamos de presentar é moi sinxelo pero este non é o caso para a gran maioría de problemas que se tratan de resolver neste campo. Para poder resolver problemas máis complexos é necesario utilizar máis tipos de capas, con diferentes funcións de activación e, como explicamos no capítulo anterior, realizar probas con distintas combinacións de hiperparámetros. Ademais, os conceptos explicados neste TFG son os básicos necesarios para comprender o funcionamento das redes neuronais, pero existen arquitecturas, funcións e métodos moito máis avanzados que os explicados que axudan ao obxectivo de resolver problemas complexos. Na seguinte sección presentaranse diferentes exemplos de redes neuronais utilizadas actualmente.

3.2. Aplicacións reais

As redes neuronais teñen aplicacións en múltiples e diversos campos. Dende procesamento de linguaxe natural ata visión por ordenador, pasando por moitos outros tipos de tarefas, a aprendizaxe profunda é unha ferramenta cunha enorme utilidade para solucionar distintos problemas.

Como dixemos anteriormente, a maioría das redes modernas empregan conceptos moito máis avanzados nos seus modelos, pero non soamente nos modelos, senón que, para resolver algúns

problemas como NLP, teñen que realizar un intenso preprocesamento dos datos para poder empregalos na rede neuronal, que só acepta entradas numéricas. Neste tipo de técnicas inclúense os vectores *one-hot* e os *embeddings*, entre outras. No campo da minería de procesos utilízase habitualmente a aprendizaxe profunda para a predición de eventos dunha secuencia, como por exemplo nas publicacións de Evermann *et al.* (2017) ou Tax *et al.* (2017). Utilízanse en moitos artigos deste campo redes recorrentes xa que son moi axeitadas para tratar secuencias, como son os procesos. Unha publicación moi novedosa neste campo é a de Camargo *et al.* (2021), que realiza este tipo de predicións comezando cunha entrada "baleira", é dicir, sen identificar ningún elemento anterior na secuencia. Esta publicación inclúe ambas técnicas de preprocesamento mencionadas, tanto os *embeddings* como os vectores *one-hot*.

As redes recorrentes tamén son moi utilizadas no campo da tradución automática (*machine translation*), como nas publicacións de Sutskever *et al.* (2014) e Bahdanau *et al.* (2015). Esta última emprega ademais unha arquitectura chamada mecanismos de atención (*attention mechanisms*) que axudan a que a rede se centre en partes concretas da entrada ao procesar grandes cantidades de información (Vaswani *et al.*, 2017), ao igual que Luong *et al.* (2015). Esta arquitectura é moi útil en diversos campos de aprendizaxe profunda (Niu *et al.*, 2021). Na publicación de Sehovac & Grolinger (2021), por exemplo, empregan unha arquitectura de redes recorrentes combinada con mecanismos de atención para previsión de carga (*load forecasting*). A continuación dos mecanismos de atención están os *transformers* que, ao igual que as redes recorrentes, procesan secuencias de datos, pero a diferencia é que procesan toda a secuencia á vez e utilizan mecanismos de atención. Un exemplo de arquitectura de *transformers* son as publicacións de Radford *et al.* (2019), que presenta o modelo GPT-2, e (Chen *et al.*, 2020). Nos últimos anos estase favorecendo o uso de *transformers* fronte a redes LSTM no campo do procesamento de linguaxe natural (Wolf *et al.*, 2020). Podemos observar unha representación da arquitectura dos *transformers* na Figura 3.4.

Durante moitos anos, a competición **ImageNet** serviu como o barómetro do progreso en aprendizaxe supervisada no campo da visión por ordenador (Zhang *et al.*, 2021), provendo unha base de datos de máis de 14 millóns de imaxes etiquetadas en máis de 1000 clases. Algúns dos gañadores ou subcampeóns deste concurso inclúen modelos como **AlexNet** (Krizhevsky *et al.*, 2017), a primeira rede de visión a gran escala; a rede **VGG** (Simonyan & Zisserman, 2015), que emprega bloques repetidos de elementos; **GoogLeNet** (Szegedy *et al.*, 2015), que usa redes con concatenacións paralelas; e as redes residuais (*residual networks*), **ResNet** (He *et al.*, 2016), que segue sendo a arquitectura estándar máis popular en visión por ordenador. Todos estes modelos empregan redes convolucionais pero cada un deles con distintas características.

Recentemente publicouse unha nova rede neuronal do campo da visión por ordenador que é capaz de crear imaxes a partir de descricións textuais, **DALL-E 2** (Ramesh *et al.*, 2022),

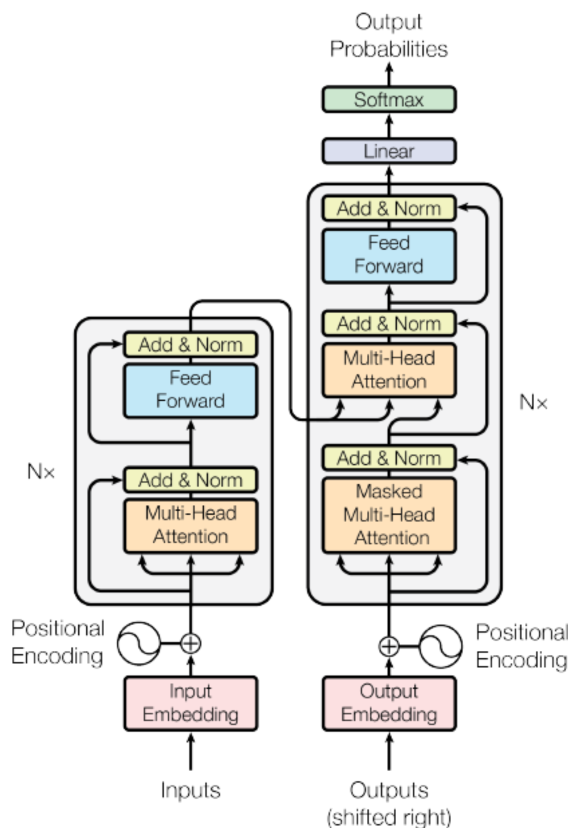


Figura 3.4: Arquitectura dun *transformer* (Vaswani *et al.*, 2017).

permitindo incluso especificar se o usuario quere que se amosen estilos como arte dixital, pintura de óleo ou renderización 3D, entre outros. Ademais permite editar imaxes para engadir elementos tendo en conta as sombras, reflexos e texturas da imaxe orixinal e crear diferentes variacións baseadas nunha imaxe orixinal. Na Figura 3.5 podemos observar dúas imaxes obtidas con este modelo a partir das descricións que as acompañan. Esta rede está baseada nas publicacións de Ramesh *et al.* (2021), Radford *et al.* (2021) e Nichol *et al.* (2021), que empregan modelos de difusión, *transformers*, *embeddings* e enormes conxuntos de datos de imaxes obtidas na *web* para obter modelos capaces de xerar imaxes a partir de texto. Ademais mestura características de modelos empregados en procesamento de linguaxe natural para a creación das imaxes a partir de texto.

a dolphin in an astronaut suit on saturn, artstation

a teddy bear on a skateboard in times square

Figura 3.5: Imaxes obtidas con DALL-E 2 a partir de descricións (Ramesh *et al.*, 2022).

Anexo I: Código

O código correspondente ao exemplo presentado no Capítulo 3 pode atoparse neste anexo, así como na seguinte ligazón: Código Python.

Para a súa execución recoméndase a utilización dun xestor de entornos, paquetes e librerías como Anaconda. É precisa a instalación das librerías **Tensorflow**, **Sklearn**, **Matplotlib**, **Math** e **Random**.

```
import math
import random
import matplotlib.pyplot as plt
import sklearn
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split

# Xeramos a semente para poder repetir o mesmo experimento
random.seed(0)
tf.random.set_seed(0)

# Inicializamos os radios
radio = 1
radio_anel1 = 1.2
radio_anel2 = 2
circulo = list()
anel = list()

# Creamos os puntos do círculo e anel
for i in range(0, 250):
```

```
    r = math.sqrt(random.uniform(0, radio * radio))
    theta = 2 * math.pi * random.random()
    circulo.append([r * math.cos(theta), r * math.sin(theta)])

    r_anel = math.sqrt(random.uniform(radio_anel1 * radio_anel1,
                                      radio_anel2 * radio_anel2))
    theta_anel = 2 * math.pi * random.random()
    anel.append([r_anel * math.cos(theta_anel),
                 r_anel * math.sin(theta_anel)])

# Amosamos a gráfica dos datos
plt.scatter(*zip(*circulo), label='círculo')
plt.scatter(*zip(*anel), label='anel')
plt.title('Datos')
plt.legend()
plt.show()

# Asignámoslle a cada dato a sua categoría correspondente
datos = list(circulo) + list(anel)
saidas = [0 for _ in range(len(circulo))] + [1 for _ in range(len(anel))]

# Dividimos os datos en adestramento e probas
x_train, x_test, y_train, y_test = train_test_split(datos, saidas,
                                                    test_size=0.2,
                                                    random_state=0,
                                                    stratify=saidas)

# Creamos o modelo lineal
lineal = Sequential()
lineal.add(Dense(5, input_dim=2))
lineal.add(Dense(1, activation='sigmoid'))
lineal.compile(optimizer='SGD', loss='binary_crossentropy',
               metrics=['accuracy'])

# E o adestramos
history_lineal = lineal.fit(x_train, y_train, epochs=200)

# Creamos o modelo relu
```

```
relu = Sequential()
relu.add(Dense(5, activation='relu', input_dim=2))
relu.add(Dense(1, activation='sigmoid'))
relu.compile(optimizer='SGD', loss='binary_crossentropy',
             metrics=['accuracy'])

# E o adestramos
history_relu = relu.fit(x_train, y_train, epochs=200)

# Creamos o modelo multicapa
multicapa = Sequential()
multicapa.add(Dense(5, activation='relu', input_dim=2))
multicapa.add(Dense(5, activation='relu'))
multicapa.add(Dense(5, activation='relu'))
multicapa.add(Dense(1, activation='sigmoid'))
multicapa.compile(optimizer='SGD', loss='binary_crossentropy',
                  metrics=['accuracy'])

# E o adestramos
history_multicapa = multicapa.fit(x_train, y_train, epochs=200)

# Amosamos a gráfica de custo
plt.plot(history_lineal.history['loss'], label='lineal')
plt.plot(history_relu.history['loss'], label='relu')
plt.plot(history_multicapa.history['loss'], label='multicapa')
plt.title('Custo do modelo')
plt.xlabel('Épocas')
plt.ylabel('Custo')
plt.legend()
plt.show()

# Amosamos a gráfica de precisión
plt.plot(history_lineal.history['accuracy'], label='lineal')
plt.plot(history_relu.history['accuracy'], label='relu')
plt.plot(history_multicapa.history['accuracy'], label='multicapa')
plt.title('Precisión do modelo')
plt.xlabel('Épocas')
plt.ylabel('Precisión')
```

```
plt.legend()
plt.show()

# Imprimimos a porcentaxe de precisión en adestramento
# para os 3 modelos
res = lineal.evaluate(x_train, y_train)
print('Lineal. Precisión en adestramento: {}%'.format(res[1]))
res_relu = relu.evaluate(x_train, y_train)
print('Relu. Precisión en adestramento: {}%'.format(res_relu[1]))
res_multicapa = multicapa.evaluate(x_train, y_train)
print('Multicapa. Precisión en adestramento: {}%'.format(res_multicapa[1]))

# Imprimimos a porcentaxe de precisión en probas
# para os 3 modelos
res2 = lineal.evaluate(x_test, y_test)
print('Lineal. Precisión en probas: {}%'.format(res2[1]))
res_relu = relu.evaluate(x_test, y_test)
print('Relu. Precisión en probas: {}%'.format(res_relu[1]))
res_multicapa = multicapa.evaluate(x_test, y_test)
print('Multicapa. Precisión en probas: {}%'.format(res_multicapa[1]))
```

Siglas

ADALINE *Adaptive Linear Element*. 6

ANN *Artificial Neural Network*. 6

CNN *Convolutional Neural Network*. 20, 21, 25

FCL *Fully-Connected Layer*. 21, 27

GPT *Generative Pre-trained Transformers*. 45

GPU *Graphics Processing Unit*. 21

GRU *Gated Recurrent Unit*. 17, 19, 20

IA Intelixencia Artificial. 3, 5

LSTM *Long Short-Term Memory*. 7, 17–19, 45

MAE *Mean Absolute Error*. 28

MLP *Multi-Layer Perceptron*. 4

MSE *Mean Squared Error*. 28

NAG *Nesterov Accelerated Gradient*. 34–36

NLP *Natural Language Processing*. 7, 45

ReLU *Rectified Linear Unit*. 7, 13, 24, 42, 43

RMS *Root Mean Squared*. 37

RNN *Recurrent Neural Network*. 15, 17, 19

SELU *Scaled Exponential Linear Units*. 13

SGD *Stochastic Gradient Descent*. 30, 31, 34, 35

TFG Traballo de Fin de Grao. 44

VGG *Visual Geometry Group*. 45

Bibliografía

Abdulwahab, S., Jabreel, M. & Moreno, A. (2017). *Deep Learning Models for Paraphrases Identification*. ResearchGate. https://doi.org/10.13140/RG.2.2.15743.46240

Aggarwal, C. C. (2018). *Neural Networks and Deep Learning*. Springer International Publishing. https://doi.org/10.1007/978-3-319-94463-0

Baccouche, M., Mamalet, F., Wolf, C., Garcia, C., & Baskurt, A. (2011). *Sequential deep learning for human action recognition*. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 7065 LNCS, pp. 29–39). https://doi.org/10.1007/978-3-642-25446-8_4

Bahdanau, D., Cho, K. H., & Bengio, Y. (2015). *Neural machine translation by jointly learning to align and translate*. In: 3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings. International Conference on Learning Representations, ICLR. https://doi.org/10.48550/arXiv.1409.0473

Banan, S., Ridwan, M., & Adisaputera, A. (2020). *A study of connectionism theory*. Budapest International Research and Critics Institute (BIRCI-Journal): Humanities and Social Sciences, 3(3), 2335–2342. https://doi.org/10.33258/birci.v3i3.1181

Barro, S. (2022). *Matemáticas que aprenden*. El Correo Gallego. https://www.elcorreogallego.es/galicia/matematicas-que-aprenden-GB10668538 (last accessed 27/06/2022).

Bayer (2019). *Machine learning: How a game of checkers is shaping agriculture*. Forbes. https://www.forbes.com/sites/bayer/2019/10/14/machine-learning-how-a-game-of-checkers-is-shaping-agriculture/?sh=1924bd984c42 (last accessed 27/06/2022).

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). *Algorithms for hyper-parameter optimization*. In: Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011, NIPS 2011. https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf

Bottou, L. (1991). *Stochastic gradient learning in neural networks.* Proceedings of Neuro-Nîmes, 91(8), 12. http://leon.bottou.org/publications/pdf/nimes-1991.pdf

Brochu, E., Cora, V. M., & De Freitas, N. (2010). *A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning.* arXiv. https://doi.org/10.48550/arXiv.1012.2599

Camargo, M., Dumas, M., & González-Rojas, O. (2021). *Discovering generative models from event logs: Data-driven simulation vs deep learning.* PeerJ Computer Science, 7, 1–23. https://doi.org/10.7717/PEERJ-CS.577

Chen, M., Radford, A., Child, R., Wu, J., Jun, H., Luan, D., & Sutskever, I. (2020). *Generative pretraining from pixels.* In: 37th International Conference on Machine Learning, ICML 2020 (Vol. PartF168147-3, pp. 1669–1681). International Machine Learning Society (IMLS). https://proceedings.mlr.press/v119/chen20s.html

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). *Learning phrase representations using RNN encoder-decoder for statistical machine translation.* arXiv. https://doi.org/10.48550/arXiv.1406.1078.

Darken, C., Chang, J., & Moody, J. (1992). *Learning rate schedules for faster stochastic gradient search.* In: Neural Networks for Signal Processing - Proceedings of the IEEE Workshop (pp. 3–12). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/NNSP.1992.253713

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (2014). *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization.* In: Advances in Neural Information Processing Systems (Vol. 4, pp. 2933–2941). Neural Information Processing Systems Foundation.

Du, J. (2019). *The frontier of SGD and its variants in machine learning.* In: Journal of Physics: Conference Series (Vol. 1229). Institute of Physics Publishing. https://doi.org/10.1088/1742-6596/1229/1/012046

Duchi, J., Hazan, E., & Singer, Y. (2011). *Adaptive subgradient methods for online learning and stochastic optimization.* Journal of Machine Learning Research, 12, 2121–2159. https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

Evermann, J., Rehse, J. R., & Fettke, P. (2017). *Predicting process behaviour using deep learning.* Decision Support Systems, 100, 129–140. https://doi.org/10.1016/j.dss.2017.04.003

Fukushima, K. (1980). *Neocognitron: a self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position.* Biological Cybernetics, 36(4), 193–202. https://doi.org/10.1007/BF00344251

Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning.* The MIT Press. http://www.deeplearningbook.org (last accessed 27/06/2022).

Graves, A. (2014). *Generating sequences with recurrent neural networks.* arXiv. https://doi.org/10.48550/arXiv.1308.0850

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2016). *LSTM: A search space odyssey.* IEEE Transactions on Neural Networks and Learning Systems, 28(10), 2222-2232. https://doi.org/10.1109/TNNLS.2016.2582924

He, K., Zhang, X., Ren, S., & Sun, J. (2016). *Deep residual learning for image recognition.* In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Vol. 2016-December, pp. 770–778). IEEE Computer Society. https://doi.org/10.1109/CVPR.2016.90

Hinton, G. E. (2007a). *To recognize shapes, first learn to generate images.* Progress in Brain Research, 165, 535–547. Elsevier. https://doi.org/10.1016/S0079-6123(06)65034-6

Hinton, G. E. (2007b). *Learning multiple layers of representation.* Trends in Cognitive Sciences, 11(10), 428-434. https://doi.org/10.1016/j.tics.2007.09.004

Hochreiter, S., & Schmidhuber, J. (1997). *Long short-term memory.* Neural Computation, 9(8), 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735

Hutter, F., Hoos, H. H. & Leyton-Brown, K. (2011). *Sequential model-based optimization for general algorithm configuration.* In: Coello, C. A. C. (Ed.) Learning and Intelligent Optimization. LION 2011. Lecture Notes in Computer Science, vol 6683. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-25566-3_40

IBM Cloud Education. (2020a). *Redes neuronales.* IBM. https://www.ibm.com/es-es/cloud/learn/neural-networks (last accessed 27/06/2022).

IBM Cloud Education (2020b). *Recurrent neural networks.* IBM. https://www.ibm.com/cloud/learn/recurrent-neural-networks (last accessed 27/06/2022).

IBM Cloud Education. (2020c). *Convolutional neural networks.* IBM. https://www.ibm.com/cloud/learn/convolutional-neural-networks (last accessed 27/06/2022).

Ji, S., Xu, W., Yang, M., & Yu, K. (2013). *3D convolutional neural networks for human action recognition.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(1), 221–231. https://doi.org/10.1109/TPAMI.2012.59

Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., & Li, F. F. (2014). *Large-scale video classification with convolutional neural networks.* In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (pp. 1725–1732). IEEE Computer Society. https://doi.org/10.1109/CVPR.2014.223

Kingma, D. P., & Ba, J. L. (2015). *Adam: A method for stochastic optimization.* In: 3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings. International Conference on Learning Representations, ICLR. https://doi.org/10.48550/arXiv.1412.6980

Kirsch, S. (2021). *What is pooling in a convolutional neural network (CNN): Pooling layers explained.* Programmathically. https://programmathically.com/what-is-pooling-in-a-convolutional-neural-network-cnn-pooling-layers-explained/ (last accessed 27/06/2022).

Kostadinov, S. (2017). *Understanding GRU networks.* Towards Data Science. https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be (last accessed 27/06/2022).

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). *ImageNet classification with deep convolutional neural networks.* Communications of the ACM, 60(6), 84–90. https://doi.org/10.1145/3065386

Kumar, A., Sarkar, S., & Pradhan, C. (2020). *Malaria disease detection using CNN technique with SGD, RMSprop and ADAM optimizers.* In: Deep Learning Techniques for Biomedical and Health Informatics (pp. 211–230). Springer. https://doi.org/10.1007/978-3-030-33966-1_11

Le Cun, Y., & Fogelman-Soulié, F. (1987). *Modèles connexionnistes de l'apprentissage.* Intellectica. Revue de l'Association pour la Recherche Cognitive, 2(1), 114–143. https://doi.org/10.3406/intel.1987.1804

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). *Backpropagation applied to handwritten zip code recognition.* Neural Computation, 1(4), 541–551. https://doi.org/10.1162/neco.1989.1.4.541

Lopez Pinaya, W. H., Vieira, S., Garcia-Dias, R., & Mechelli, A. (2019). *Convolutional neural networks.* In: Machine Learning: Methods and Applications to Brain Disorders (pp. 173-191). Mechelli, A. & Vieira, S. (Eds.). Elsevier. https://doi.org/10.1016/B978-0-12-815739-8.00010-9

Lucas, Y. (2019). *Credit card fraud detection using machine learning with integration of contextual knowledge*. Université de Lyon; Universität Passau (Deutscheland). Retrieved from http://theses.insa-lyon.fr/publication/2019LYSEI110/these.pdf (last accessed 27/06/2022).

Luong, M. T., Pham, H., & Manning, C. D. (2015). *Effective approaches to attention-based neural machine translation*. In: Conference Proceedings - EMNLP 2015: Conference on Empirical Methods in Natural Language Processing (pp. 1412–1421). Association for Computational Linguistics (ACL). https://doi.org/10.18653/v1/d15-1166

McCulloch, W. S., & Pitts, W. (1943). *A logical calculus of the ideas immanent in nervous activity*. The Bulletin of Mathematical Biophysics, 5(4), 115–133. https://doi.org/10.1007/BF02478259

Nesterov, Y. (1983). *A method for unconstrained convex minimization problem with the rate of convergence* $o(1/k^2)$. Doklady AN USSR, 269, 543–547. Retrieved from https://ci.nii.ac.jp/naid/20001173129/en/ (last accessed 27/06/2022).

Nichol, A., Dhariwal, P., Ramesh, A., Shyam, P., Mishkin, P., McGrew, B., ... & Chen, M. (2021). *Glide: Towards photorealistic image generation and editing with text-guided diffusion models*. arXiv. https://doi.org/10.48550/arXiv.2112.10741

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press. http://neuralnetworksanddeeplearning.com/ (last accessed 30/06/2022).

Niu, Z., Zhong, G., & Yu, H. (2021). *A review on the attention mechanism of deep learning*. Neurocomputing, 452, 48–62. https://doi.org/10.1016/j.neucom.2021.03.091

Olah, C. (2015). *Understanding LSTM networks*. Colah's Blog. http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (last accessed 27/06/2022).

Panton, K., Matuszek, C., Lenat, D., Schneider, D., Witbrock, M., Siegel, N., & Shepard, B. (2006). *Common sense reasoning - From cyc to intelligent assistant*. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 3864 LNAI, pp. 1–31). https://doi.org/10.1007/11825890_1

Powers, D. M. (2020). *Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation*. arXiv. https://doi.org/10.48550/arXiv.2010.16061

Qian, N. (1999). *On the momentum term in gradient descent learning algorithms*. Neural Networks, 12(1), 145–151. https://doi.org/10.1016/S0893-6080(98)00116-6

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). *Language models are unsupervised multitask learners*. OpenAI blog, 1(8), 9.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., ... & Sutskever, I. (2021). *Learning transferable visual models from natural language supervision*. In: International Conference on Machine Learning (pp. 8748-8763). PMLR. https://doi.org/10.48550/arXiv.2103.00020

Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., ... & Sutskever, I. (2021). *Zero-shot text-to-image generation*. In: International Conference on Machine Learning (pp. 8821-8831). PMLR. https://doi.org/10.48550/arXiv.2102.12092

Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., & Chen, M. (2022). *Hierarchical text-conditional image generation with clip latents*. arXiv. https://doi.org/10.48550/arXiv.2204.06125

Rasamoelina, A. D., Adjailia, F. & Sincak, P. (2020). *A review of activation function for artificial neural network*. In *SAMI 2020 - IEEE 18th World Symposium on Applied Machine Intelligence and Informatics, Proceedings* (pp. 281–286). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/SAMI48414.2020.9108717

Robbins, H., & Monro, S. (1951). *A stochastic approximation method*. The Annals of Mathematical Statistics, 22(3), 400-407. https://doi.org/10.1214/aoms/1177729586

Rosenblatt, F. (1958). *The perceptron: A probabilistic model for information storage and organization in the brain*. Psychological Review, 65(6), 386–408. https://doi.org/10.1037/h0042519

Rosenblatt, F. (1960). *Perceptron simulation experiments*. Proceedings of the IRE, 48(3), 301–309. https://doi.org/10.1109/JRPROC.1960.287598

Ruder, S. (2016). *An overview of gradient descent optimization algorithms*. arXiv. https://doi.org/10.48550/arXiv.1609.04747

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). *Learning representations by back-propagating errors*. Nature, 323(6088), 533-536. https://doi.org/10.1038/323533a0

Russell, S. J., & Norvig, P. (2016). *Artificial Neural Networks*. In: Artificial Intelligence: A Modern Approach (Third Edition, pp. 727–737). Pearson.

Sehovac, L., & Grolinger, K. (2021). *Deep learning for load forecasting: Sequence to sequence recurrent neural networks with attention*. IEEE Access, 8, 36411–36426. https://doi.org/10.1109/ACCESS.2020.2975738

Simonyan, K., & Zisserman, A. (2014). *Two-stream convolutional networks for action recognition in videos*. In: Advances in Neural Information Processing Systems (Vol. 1, pp. 568–576). Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. & Weinberger, K. Q. (Eds.). Neural Information Processing Systems Foundation. https://proceedings.neurips.cc/paper/2014/file/00ec53c4682d36f5c4359f4ae7bd7ba1-Paper.pdf

Simonyan, K., & Zisserman, A. (2015). *Very deep convolutional networks for large-scale image recognition.* In: 3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings. International Conference on Learning Representations, ICLR. https://doi.org/10.48550/arXiv.1409.1556

Snoek, J., Larochelle, H., & Adams, R. P. (2012). *Practical Bayesian optimization of machine learning algorithms.* In: Advances in Neural Information Processing Systems (Vol. 4, pp. 2951–2959). Pereira, F., Burges, C. J., Bottou, L. & Weinberger, K. Q. (Eds.). Neural Information Processing Systems Foundation. https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). *Sequence to sequence learning with neural networks.* In: Advances in Neural Information Processing Systems (Vol. 4, pp. 3104–3112). Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. & Weinberger, K. Q. (Eds.). Neural Information Processing Systems Foundation. https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf

Sutton, R. S. (1986). *Two problems with backpropagation and other steepest-descent learning procedures for networks.* In: Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986 (pp. 823-832). https://cir.nii.ac.jp/crid/1572824499995923584

Swade, D. (2000). *The cogwheel brain: Charles Babbage and the quest to build the first computer.* Little, Brown and Company.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). *Going deeper with convolutions.* In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Vol. 07-12-June-2015, pp. 1–9). IEEE Computer Society. https://doi.org/10.1109/CVPR.2015.7298594

Tax, N., Verenich, I., La Rosa, M., & Dumas, M. (2017). *Predictive business process monitoring with LSTM neural networks.* In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 10253 LNCS, pp. 477–492). Springer Verlag. https://doi.org/10.1007/978-3-319-59536-8_30

Varsamopoulos, S., Bertels, K. & Almudever, C. (2018). *Designing neural network based decoders for surface codes.* arXiv. https://doi.org/10.48550/arXiv.1811.12456

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). *Attention is all you need.* In: Advances in Neural Information Processing Systems (Vol. 2017-December, pp. 5999–6009). Neural Information Processing Systems Foundation.

Wang, Z., Li, K., Xia, S. Q., & Liu, H. (2021). *Economic recession prediction using deep neural network.* arXiv. https://doi.org/10.48550/arXiv.2107.10980

Webb, G. I., Keogh, E., & Miikkulainen, R. (2010). *Naïve Bayes*. Encyclopedia of Machine Learning, 15, 713-714.

Werbos, P. J. (1974). *Beyond Regression : New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University.

Werbos, P. J. (1990). *Backpropagation through time: What it does and how to do it*. Proceedings of the IEEE, 78(10), 1550–1560. `https://doi.org/10.1109/5.58337`

Widrow, B., & Hoff, M. E. (1960). *Adaptive switching circuits*. Stanford University California Stanford Electronics Labs.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., ... & Rush, A. (2020). *Transformers: State-of-the-art natural language processing* (pp. 38–45). Association for Computational Linguistics (ACL). *https://doi.org/10.18653/v1/2020.emnlp-demos.6*

Zakeri Nasrabadi, M., Parsa, S., & Kalaee, A. (2021). *Format-aware learn&fuzz: deep test data generation for efficient fuzzing*. Neural Computing and Applications, 33(5), 1497–1513. `https://doi.org/10.1007/s00521-020-05039-7`

Zhang, Z., & Sabuncu, M. R. (2018). *Generalized cross entropy loss for training deep neural networks with noisy labels*. In: Advances in Neural Information Processing Systems (Vol. 2018-December, pp. 8778–8788). Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N. & Garnett, R. (Eds.). Neural Information Processing Systems Foundation. `https://proceedings.neurips.cc/paper/2018/file/f2925f97bc13ad2852a7a551802feea0-Paper.pdf`

Zhang, A., Lipton, Z. C., Li, M. & Smola, A. J. (2021). *Dive into Deep Learning*. arXiv. `https://doi.org/10.48550/arxiv.2106.11342`